

# **A little book on the custom OS development from scratch**

Kwangdo Yi

# Contents

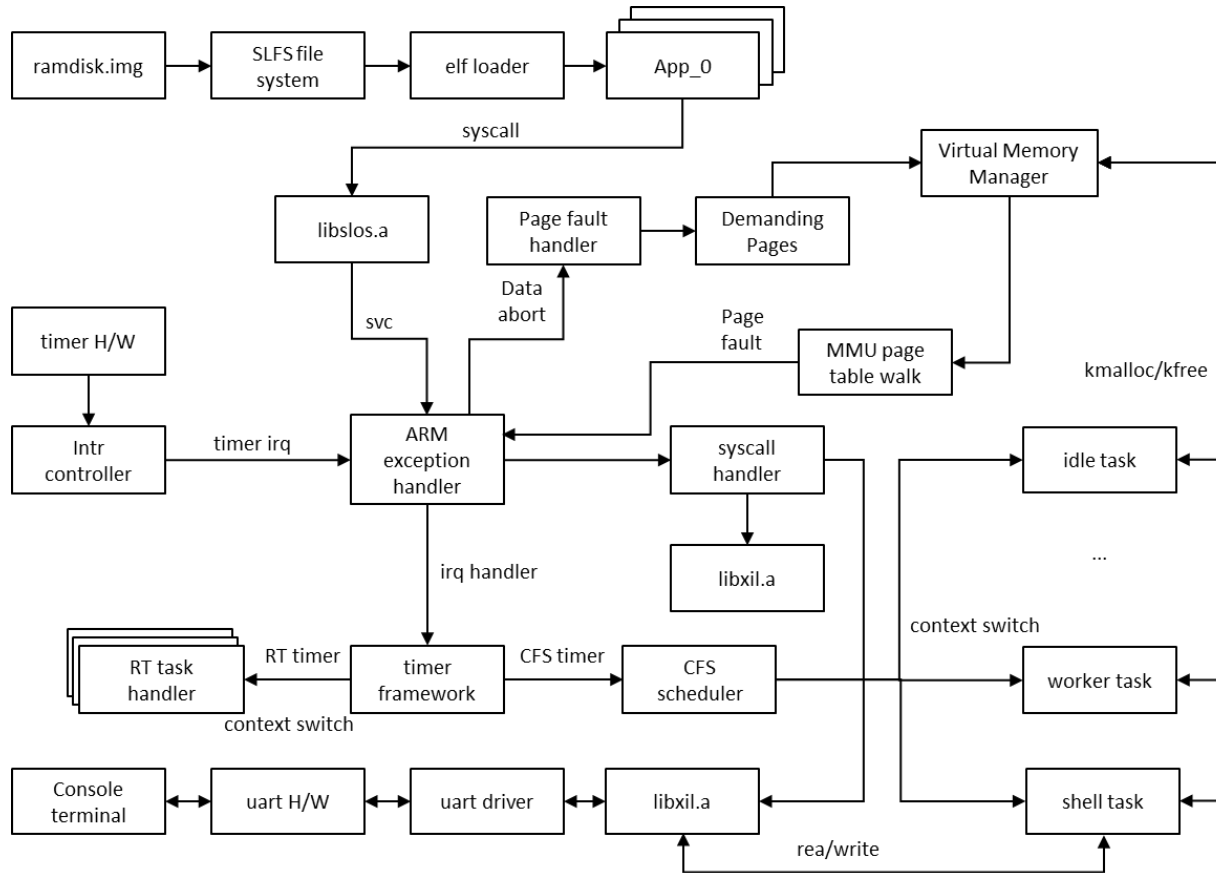
1	Introduction .....	6
2	Environment Setup .....	7
2.1	Development Environment Overview .....	7
2.2	ARM Introduction .....	8
2.2.1	ARM Processor Profile .....	8
2.2.2	Architecture Extension.....	8
2.2.3	ARM ISA Overview .....	9
2.2.4	ARM Processor Modes .....	11
2.2.5	ARM Core Registers and Program Status Registers(PSRs) .....	12
2.2.6	Security States.....	18
2.2.7	Privilege Levels.....	18
2.2.8	Relationship among Mode, Privilege, and Security .....	19
2.2.9	Cortex-A9 and Cortex-A9 MPCore .....	20
2.3	Zynq-7000 Introduction .....	20
2.4	Development PC Setup .....	22
2.4.1	Ubuntu 16.04 Installation .....	22
2.4.2	Git Installation.....	22
2.4.3	Serial Terminal Installation .....	23
2.4.4	GCC Tool Chain Installation.....	25
2.4.5	Miscellaneous Settings.....	26
2.5	Development Board Setup.....	27
2.6	Xilinx SDK, Vivado, and Petalinux Installation.....	28
2.6.1	Petalinux Installation.....	28
2.6.2	Vivado Installatioin, Default Block Design, and Xilinx SDK.....	28
3	Building Sources and SLOS Boot Up in Zynq7000 .....	31
3.1	Base SLOS Sources .....	31
3.2	Linker Script.....	32
3.2.1	Linker Script Basics.....	33
3.2.2	Simple Linker Script.....	33
3.3	Zynq7000 booting sequence.....	36

3.4	Exception Vector and Reset Handler .....	37
3.5	Kernel Main Entry Point .....	39
3.6	Porting Xilinx UART Drivers .....	39
3.7	Makefile and SLOS Build Process .....	40
3.7.1	Overview of Makefile .....	40
3.7.2	SLOS build process .....	41
3.7.3	Bootting SLOS in The Development Board .....	44
3.8	Debugger in SLOS .....	45
3.8.1	How to debugging .....	45
3.8.2	XMD and GDB Commands .....	46
4	Process Management .....	52
4.1	Exception Handler .....	52
4.1.1	System Level Interrupt .....	52
4.1.2	Reset Handler and IRQ Handler .....	54
4.1.3	Reset Handler .....	57
4.1.4	IRQ Handler in SLOS .....	58
4.2	Generic Interrupt Controller (GIC) Implementation .....	59
4.2.1	GIC Partitioning .....	59
4.2.2	Interrupt State and Handling Sequence.....	59
4.2.3	GIC Implementation .....	61
4.3	Timer Framework.....	62
4.3.1	Timer Interrupt Service Routine Implementation .....	63
4.3.2	Quick Review on Linux Timer Framework.....	65
4.3.3	SLOS Timer Framework Implementation.....	65
4.3.4	SLOS Timer Framework Test .....	70
4.4	SLOS Process Management.....	71
4.4.1	Task Control Block.....	71
4.4.2	Task Creation.....	72
4.4.3	Task State .....	74
4.4.4	Context Switching .....	75
4.4.5	Task Synchronization .....	77
4.5	CFS Scheduler.....	80

4.5.1	Run Queue, Sched Entity, vruntime, jiffies .....	80
4.5.2	schedule .....	81
4.5.3	Test of CFS Scheduler .....	83
4.6	Realtime Scheduler .....	84
4.6.1	yield() .....	84
4.6.2	Test Realtime Scheduler .....	85
4.7	Putting it altogether .....	86
4.7.1	Kernel Main Function Sequence .....	86
4.7.2	CPU Idle Process and Dynamic Power Management.....	82
5	Memory Management.....	89
5.1	Logical Address, Virtual Address and Physical Address .....	89
5.2	Memory Management Source Tree .....	89
5.3	ARM MMU description .....	90
5.3.1	Address Translation in ARM MMU .....	90
5.3.2	Small Page Table Walk .....	91
5.4	SLOS Virtual Memory .....	91
5.4.1	Linker Script Update.....	91
5.4.2	SLOS Virtual Memory Map.....	93
5.5	Two Stage Address Translation in SLOS .....	94
5.5.1	Page Translation Table Initialization .....	94
5.5.2	Initialization of First Stage Translation Table.....	95
5.5.3	Initialization of Second Stage Translation Table .....	97
5.5.4	MMU Programming .....	98
5.6	Memory Manager in SLOS .....	100
5.6.1	Kernel Frame .....	101
5.6.2	Virtual Memory Pool .....	102
5.7	Demanding Page Implementation .....	104
5.7.1	Fault Checking Sequence .....	104
5.7.2	Data Abort Handler .....	105
5.7.3	Page Fault Handling .....	107
5.7.4	Demanding Page Flow.....	108
5.8	Putting It Altogether .....	108

6	Storage Management .....	112
6.1	SLFS System.....	112
6.1.1	Design Architecture.....	112
6.1.2	SLFS Source Tree .....	112
6.2	File System .....	113
6.2.1	Description of SLFS.....	113
6.2.2	inode for SLFS.....	113
6.2.3	Mount and Format SLFS.....	114
6.2.4	2 Level Data Block Indexing .....	114
6.3	File in SLFS.....	115
6.3.1	Ramdisk I/O.....	115
6.3.2	File Open, File Close and File Delete .....	116
6.3.3	File Read.....	117
6.3.4	File Write.....	118
6.3.5	File Read/Write Test .....	119
6.4	System Call .....	121
6.4.1	SVC Handler Implementation .....	121
6.4.2	Syscall Library for User Application.....	123
6.5	HelloWorld Application and Ramdisk .....	124
6.5.1	HelloWorld Application.....	124
6.5.2	Building ramdisk.img.....	124
6.5.3	Loading ramdisk.img from fsbl.....	125
6.6	ELF Loader for User Application.....	126
6.6.1	Creation Application Files .....	127
6.6.2	Elf .....	128
6.6.3	Loading User Application ELF.....	128
6.7	Putting It Altogether .....	132
6.7.1	Changes in Build Process.....	132
6.7.2	Let's see it .....	132
7	Design a Custom HW in FPGA and Device Driver .....	135

# 1 Introduction



## 2 Environment Setup

### 2.1 Development Environment Overview

Since operating system is running right on top of hardware and interfacing software and hardware, we need to have a hardware for the operating system to work on. To demonstrate, process management needs a processor which runs each processes, memory management needs MMU, and file system needs a storage such as Hard Disk. There are many commercial boards you can use for your custom operating system like Raspberry PI. When I first started to develop custom operating system, I reused my company's development board which has Qualcomm chipset, and also I could use android bootloader and downloader(fastboot). By using verified hardware, bootloader, and downloader, I don't have to waste my time working on hardware debugging or bootloader. Those are not operating system stuff but must be solved before starting to OS development. It had also well-working JTAG debugger which is very important in this type of software development.

This custom operating system will be running on embedded system, and we first have to select the proper development board. I will describe the implementations of custom OS development based on Zynq-7000 evaluation kit. You can choose different development board having ARM processor if you are able to use JTAG debugging, downloader, basic bsp(UART, timer interrupt), and bootloader which initialize hardware before loading your OS. Steps and implementation in this book also might be applied to your board with a little modification.

Development PC must have cross-compiler for ARM processor. I am using GCC(GNU Cross Compiler) tool chain running on Ubuntu 16.04 for this development. There are GCC tool chain running on Windows, but I haven't used it in Windows and I recommend GCC tool chain with Ubuntu.

Ubuntu can be installed to virtual machine. I am using Virtualbox and it is very stable and I don't experience any problems in developing OS so far. You might prefer to stand alone Ubuntu PC from Windows/Ubuntu dual boot rather than virtual machine. This is my first development environment and I didn't have any problem with that. Now that Virtualbox is very stable now and convenient to use, I recommend you Virtualbox with Ubuntu 16.04 installation.

Xilinx development tools such as Vivado, petalinux SDK are very heavy and consumes a large amount of memory and CPU time. If you are not interested in FPGA and don't want to design custom hardware and develop its device driver, you can ignore these. You can use low-tier laptop for this development. But if you want to, I recommend mid or high-tier PC.

Preparation of development board and PC are the first step to start custom operating system development, and let's dive into them more detail.

## 2.2 ARM Introduction

This book is not about the ARM processor or zynq7000 but it is the first step in operating system development to know about the hardware which the operating system will drive.

### 2.2.1 ARM Processor Profile

The ARM architecture has evolved significantly since its introduction, and ARM continues to develop it. Seven major versions of the architecture have been defined to date, denoted by the version numbers 1 to 7. Of these, the first three versions are now obsolete. ARMv7 provides three profiles:

**ARMv7-A:** Application profile, described in this manual:

- Implements a traditional ARM architecture with multiple modes.
- Supports a Virtual Memory System Architecture (VMSA) based on a *Memory Management Unit* (MMU). An ARMv7-A implementation can be called a VMSAv7 implementation.
- Supports the ARM and Thumb instruction sets.

**ARMv7-R:** Real-time profile, described in this manual:

- Implements a traditional ARM architecture with multiple modes.
- Supports a *Protected Memory System Architecture* (PMSA) based on a *Memory Protection Unit* (MPU). An ARMv7-R implementation can be called a PMSAv7 implementation.
- Supports the ARM and Thumb instruction sets.

**ARMv7-M:** Microcontroller profile, described in the *ARMv7-M Architecture Reference Manual*:

- Implements a programmers' model designed for low-latency interrupt processing, with hardware stacking of registers and support for writing interrupt handlers in high-level languages.
- Implements a variant of the ARMv7 PMSA.
- Supports a variant of the Thumb instruction set.

### 2.2.2 Architecture Extension

Instruction Set Architecture extensions are:

- **Jazelle** Is the Java bytecode execution extension that extended ARMv5TE to ARMv5TEJ. From ARMv6, the architecture requires at least the trivial Jazelle implementation, but a Jazelle implementation is still often described as a Jazelle extension. The Virtualization Extensions require that the Jazelle implementation is the trivial Jazelle implementation.
- **ThumbEE** Is an extension that provides the ThumbEE instruction set, a variant of the Thumb instruction set that is designed as a target for dynamically generated code. In the original release of the ARMv7 architecture, the ThumbEE extension was:
  - A required extension to the ARMv7-A profile.
  - An optional extension to the ARMv7-R profile.



- From publication of issue C.a of this manual, ARM deprecates any use of ThumbEE instructions. However, ARMv7-A implementations must continue to include ThumbEE support, for backwards compatibility.
- **Floating-point** Is a floating-point coprocessor extension to the instruction set architectures. For historic reasons, the Floating-point Extension is also called the *VFP Extension*. There have been the following versions of the Floating-point (VFP) Extension
- **Advanced SIMD** Is an instruction set extension that provides Single Instruction Multiple Data (SIMD) integer and single-precision floating-point vector operations on doubleword and quadword registers. There have been the following versions of Advanced

Architecture Extensions are:

- **Security Extensions:** Are an optional set of extensions to VMSAv6 implementations of the ARMv6K architecture, and to the ARMv7-A architecture profile, that provide a set of security features that facilitate the development of secure applications.
- **Multiprocessing Extensions:** Are an optional set of extensions to the ARMv7-A and ARMv7-R profiles, that provides a set of features that enhance multiprocessing functionality.
- **Large Physical Address Extension:** Is an OPTIONAL extension to VMSAv7 that provides an address translation system supporting physical addresses of up to 40 bits at a fine grain of translation. The Large Physical Address Extension requires implementation of the Multiprocessing Extensions.
- **Virtualization Extensions:** Are an optional set of extensions to VMSAv7 that provides hardware support for virtualizing the Non-secure state of a VMSAv7 implementation. This supports system use of a virtual machine monitor, also called a hypervisor, to switch Guest operating systems.
- **Generic Timer Extension:** Is an optional extension to any ARMv7-A or ARMv7-R, that provides a system timer, and a low-latency register interface to it.
- Performance Monitors Extension

### 2.2.3 ARM ISA Overview

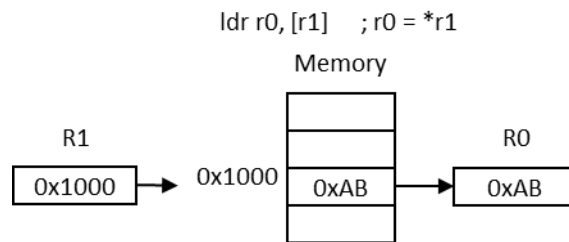
ARMv7 Architecture defines its *Instruction Set Architecture*(ISA). Most ARMv7 instructions are categorized as one of Data Processing Instruction or Load/Store Instruction, Branch Instruction.

- **Data Processing Instructions:** used for processing data in register such as add, subtract, shift, bit operations, and logical operations. Before processing data, ARM first needs to load that data from memory to register.
- **Load/Store Instructions:** used for access to memory. ARM follows load/store architecture. That means memory access operations(load/store) is separated from data processing instructions. ARM also support multiple load/store operations in one instruction. Load/Store instructions are ldr, str, ldm, stm.
- **Branch Instructions:** used for branch to target address such as B, BL, BLX.

ARM Technical Reference Manual chapter A8.8 describes the ARM assembler instructions in alphabetical order. Refer that chapter while reading ARM assembler code.

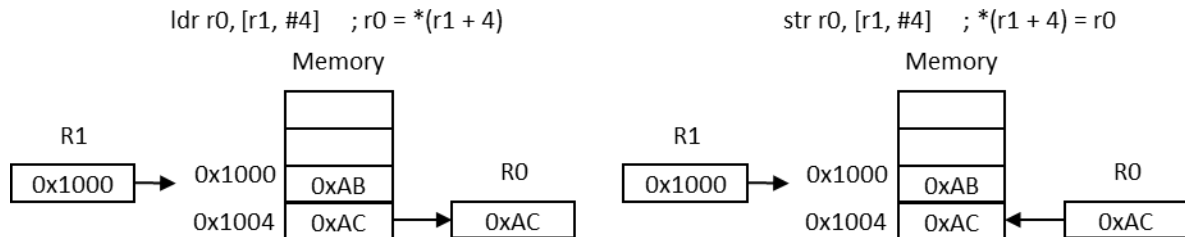
ARM is called RISC (Reduced Instruction Set Computer) processor and one of important feature of RISC processor is that arithmetic operation is clearly separated from load/store operation. That is, before running any arithmetic operations (add, subtract, etc.), the data must be loaded into register. After completion of arithmetic operation, the result need to be stored into memory again. To the contrary, there is a CISC (Complex Instruction Set Computer) which has mixed arithmetic and memory access instruction. Even CISC doesn't have fixed instruction length. A famous CISC processor is Intel processor dominantly used in desktop computer and server computer.

We need to know in detail about ARM load / store instruction. The simplest one is load a memory data to a register.

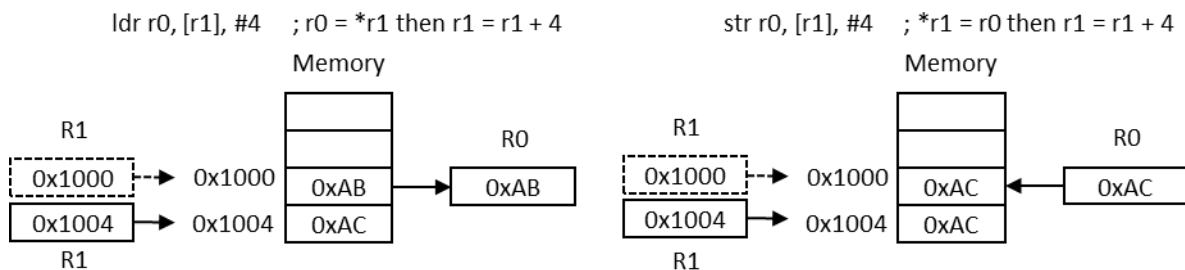


There are two more load/store forms based on addressing modes.

- Pre-indexing: The memory address is updated first and load/store the data of that memory address to/from the register.



- Post-indexing: The data in the memory is loaded/stored to/from register and the second operand is updated.



There are variations with "!" mark which represents a write-back of the register. For example, ldr r0, [r1, r2]! means r0 = \*(r1 + r2) and then r1 = r1 + r2. This write-back mark frequently used with stack pointer.

Another thing to know about load/store is about ldm/stm instructions for multiple load/store. These instructions can load/store a memory block at one instruction. They have four operation modes.

- IA: Increment address After each transfer
- IB: Increment address Before each transfer
- DA: Decrement address After each transfer
- DB: Decrement address Before each transfer

With these 4 modes, there are ldmia, ldmiB, ldmda, ldmdB, stmia, stmiB, stmda and stmdB instructions.

One interesting feature in ARM instruction set is that all ARM instructions support conditional execution. Below is condition code and its suffix used in ARM assembler instruction.

Suffix	Cond. bits	Cond. flags	Meaning
EQ	0000	Z = 1	Equal
NE	0001	Z = 0	Not equal
CS or HS	0010	C = 1	Higher or same, unsigned
CC or LO	0011	C = 0	Lower, unsigned
MI	0100	N = 1	Negative
PL	0101	N = 0	Positive or zero
VS	0110	V = 1	Overflow
VC	0111	V = 0	No overflow
HI	1000	C = 1 and Z = 0	Higher, unsigned
LS	1001	C = 0 and Z = 1	Lower or same, unsigned
GE	1010	N = V	Greater than or equal, unsigned
LT	1011	N != V	Less than, signed
GT	1100	Z = 0 and N = V	Greater than, signed
LE	1101	Z = 1 and N != V	Less than or equal, signed
AL	1110	Can have any value	Always. This is the default when no suffix is specified

Below example shows the use of a conditional instruction to find the absolute value of a number. R0 = abs(R1)

```

MOVS    R0, R1        ; R0 = R1, setting flags
IT       MI            ; skipping next instruction if value 0 or positive
RSBMI   R0, R0, #0     ; If negative, R0 = -R0

```

## 2.2.4 ARM Processor Modes

ARMv7 processor exception mode is

- User mode is the usual ARM program execution state, and is used for executing most application programs. CPSR mode bit value is 0x10.
- *Fast interrupt* (FIQ) mode is used for handling fast interrupts. CPSR mode bit value is 0x11.

- *Interrupt* (IRQ) mode is used for general-purpose interrupt handling. CPSR mode bit value is 0x12.
- Supervisor(SVC) mode is a protected mode for the OS. CPSR mode bit value is 0x13
- Abort(ABT) mode is entered after a data abort or prefetch abort. CPSR mode bit value is 0x17.
- System(SYS) mode is a privileged user mode for the OS. CPSR mode bit value is 0x1f
- Undefined(UND) mode is entered when an Undefined Instruction exception occurs. CPSR mode bit is 0x1b.
- Monitor(MON) mode is a Secure mode for the Security Extensions Secure Monitor code. CPSR mode bit is 0x16.

Modes other than User mode are collectively known as privileged modes. Privileged modes are used to service interrupts or exceptions, or to access protected resources such as interrupt enable/disable bits. Below table shows the mode structure for the processor.

### 2.2.5 ARM Core Registers and Program Status Registers(PSRs)

ARM core registers are:

- Thirteen general purpose registers, R0 to R12, that software can use for processing
- SP, the stack pointer, that can also be referred to as R13
- LR, the link register, that can also be referred to as R14
- PC, the program counter, that can also be referred to as R15.

SP, LR and PC are used for special purpose. SP is used as a pointer to the end of current frame's stack, and is often used with load/store instructions. Stack is a small memory region which is used for special purpose such as subroutine function calls, or function's local variable storage. Stack is necessary for the processor to store temporary variables while running a program. In high level application programming, we don't much care about this but in operating system development, we need to use the stack and set the stack memory for each ARM mode. In addition to load/store modes described in chapter ???, ARM stack can have 4 more modes (*Full/Empty, Ascending/Descending*). So, all modes that stack can have are as below.

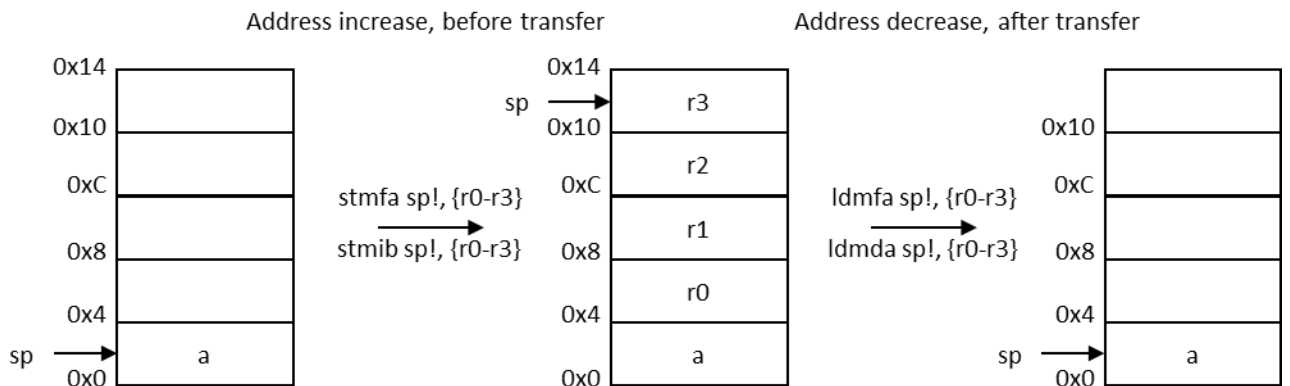
- IA: Increment address After each transfer
- IB: Increment address Before each transfer
- DA: Decrement address After each transfer
- DB: Decrement address Before each transfer
- FD: Full Descending stack,  
address decreases in store command and increases in load command,  
current stack points to an address with valid data(full),  
same with DB in store command and same with IA in load command
- ED: Empty Descending stack,  
address decreases in store command and increases in load command,  
current stack points to an empty address,  
same with DA in store command and same with IB in load command
- FA: Full Ascending stack,  
address increases in store command and decreases in load command,

current stack points to an address with valid data(full),  
same with IB in store command and same with DA in load command

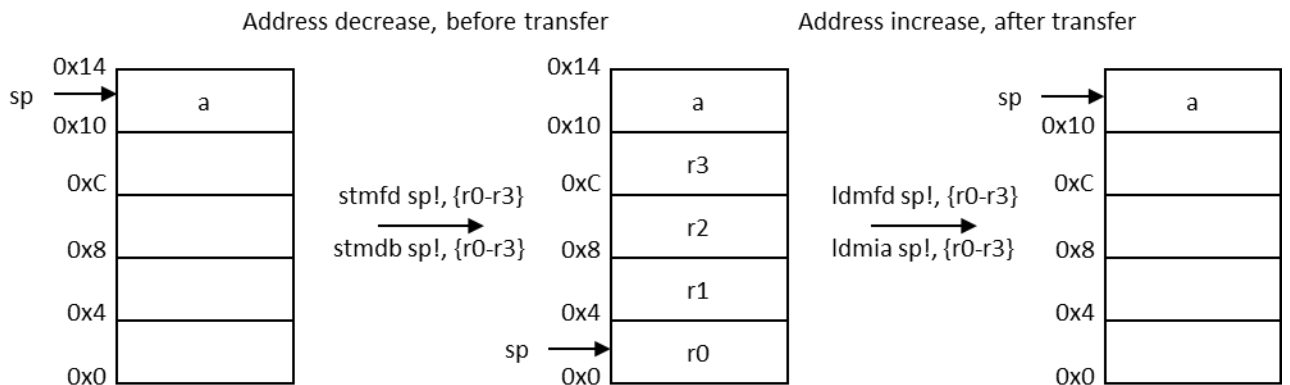
- EA: Empty Ascending stack,  
address increases in store command and decreases in load command,  
current stack points to an empty address,  
same with IA in store command and same with DB in load command

If address is increased in each transfer, load/store is in *Ascending* mode. If address is decreased, Descending mode is used. Also, there are two more cases based on the time of address change *before* or *after* load/store transfer.

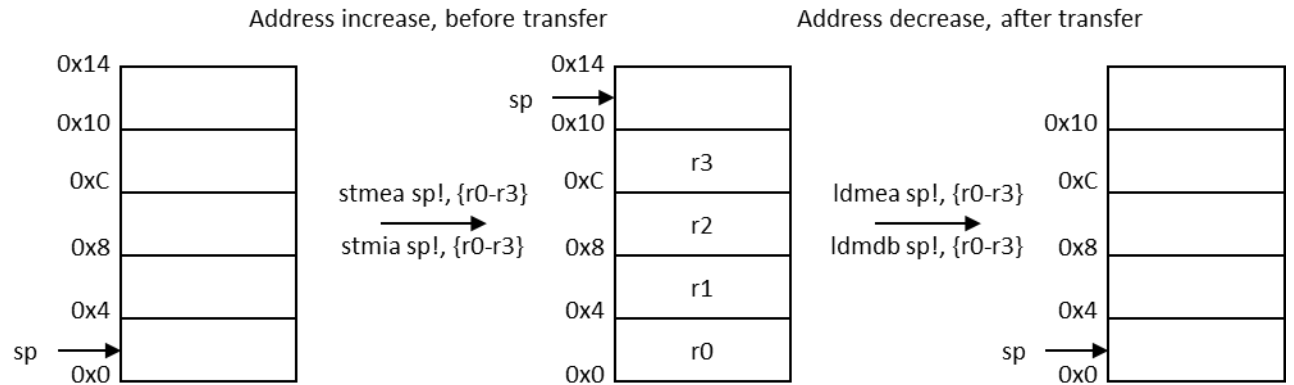
#### Full, Ascending mode stack



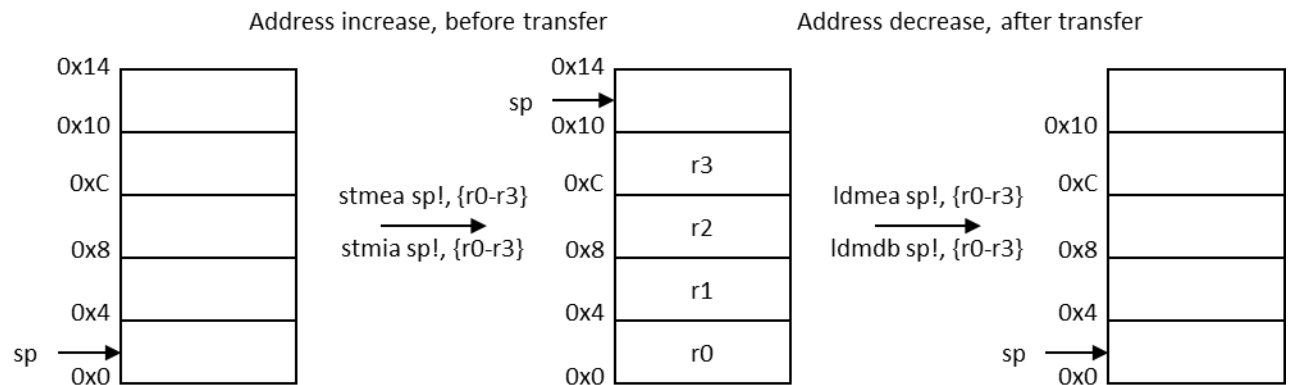
#### Full, Descending mode stack



Empty, Ascending mode stack



Empty, Ascending mode stack



LR holds return link address. PC reads as the address of current instruction plus 8 when executing an ARM instruction. This is because ARM has a pipeline of instruction and fetched instruction is current executing instruction plus 8.

R0 to R12 are used for general purpose but when execution flow jumps to other function, there is a procedure call standard for these registers. *Caller* function is for current running function and *Callee* function is a subroutine function of current function. The caller function routine must save R0~R3 to the stack and can set its argument to these registers when it calls the subroutine function. The callee function can scratch R0~R3 for its own usage and save its return value to R0. R4~R11 are callee saved registers. Especially R4~R8 are used local variables and must be remain same before and after calling the subroutine function. The subroutine function (callee function) must preserve or restore these register values when it returns to the caller function.

R11 can be used as a frame pointer. Frame pointer points to the start location of current function in stack. It is different with stack pointer which points to the end of stack. Stack pointer varies as the stack is growing, but frame pointer stays same before jumping to another subroutine function. So, frame pointer is updated only when calling another function. Since frame pointer has the start location of current function, we can use it to save function's meta information (frame information) such as pc, lr, sp, and fp.

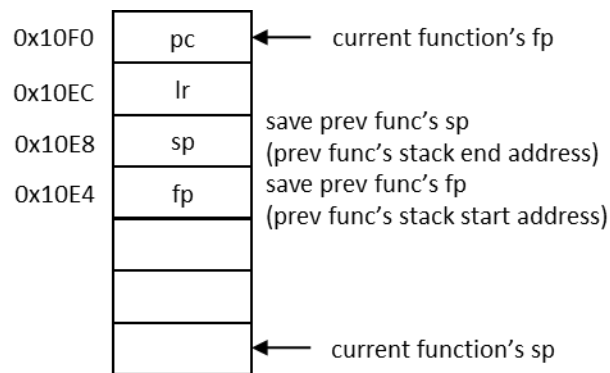
With these frame information, we can easily back trace the function call chains. In the below example, the callee function saves its frame information to the start location of its stack and use those saved information when return to caller function. Please notice that fp and sp information comes from caller function (previous function), and lr, pc are from callee function (current function). We can choose whether frame pointer is used or not by compiler option. In order to use frame pointer, ARM compiler has -fno-omit-frame-pointer, and to disable frame pointer, it has -fomit-frame-pointer option.

funcA:

```

mov    ip, sp
stmfd  sp!, {fp, ip, lr, pc}    ; save current func's lr, pc and the prev func's sp, fp
sub    fp, ip, #4               ; update fp with current func's start location in stack
/* do something here */
sub    sp, fp, #12              ; prepare to return by using current func's fp
ldmfd  sp, {fp, sp, pc}         ; return to prev func by using current func's fp

```



Below table is the summary of ARM core registers.

Register Number	Alternative Register Name	ATPCS (ARM-Thumb Procedure Call Standard) register usage
R0	a1	Caller saved registers. 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	Callee saved registers. General variable registers. The function must preserve the callee values of these registers.
R5	v2	
R6	v3	
R7	v4	
R8	v5	
R9	v6 sb	Callee saved registers. 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.





- These mask bits are set only at privilege level 1 or higher. Their values can be read in privilege level 0, but can be written only level 1 or higher. Privilege level will be explained in chapter ????. While operation system development, we need to set these mask bits.
- T, bit[5]: Thumb execution state bit. This bit and the J execution state bit, bit[24], determine the instruction set state of the processor, ARM, Thumb, Jazelle, or ThumbEE.
- M[4:0], bits[4:0]: Mode field. This field determines the current mode of processor. Below is the summary of mode bits.

Mode	Source	M[4:0]	Symbol	Purpose
User	-	0x10	USR	Normal user program execution mode.
FIQ	FIQ	0x11	FIQ	Fast interrupt mode.
IRQ	IRQ	0x12	IRQ	Interrupt mode.
Supervisor	SWI, RESET	0x13	SVC	Protected/Privileged mode for operating system
Abort	Prefetch Abort, Data Abort	0x17	ABT	Virtual memory system uses this mode for demanding page.
Undefined	Undefined Instruction	0x1b	UND	Undefined instruction execution occurred.
System	-	0x1f	SYS	Run privileged user system tasks mode. This mode has same registers with usr mode.
Hyp		0x1a	HYP	
Monitor		0x16	MON	

Some of ARM registers are banked with each mode. *Banked Register* is a register that has multiple instances, with the instance that is in use depending on the processor mode, security state, or other processor state [ARMv7 arm]. Below table describes the banked ARM core registers.

User							
and							
System	Hyp	Supervisor	Abort	Undefined	Monitor	IRQ	FIQ
r0	r0	r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7	r7	r7
r8_usr	r8	r8	r8	r8	r8	r8	r8_fiq
r9_usr	r9	r9	r9	r9	r9	r9	r9_fiq
r10_usr	r10	r10	r10	r10	r10	r10	r10_fiq
r11_usr	r11	r11	r11	r11	r11	r11	r11_fiq
r12_usr	r12	r12	r12	r12	r12	r12	r12_fiq

sp_usr	sp_hyp	sp_svc	sp_abt	sp_und	sp_mon	sp_irq	sp_fiq
lr_usr	lr_hyp	lr_svc	lr_abt	lr_und	lr_mon	lr_irq	lr_fiq
pc	pc	pc	pc	pc	pc	pc	pc

cpsr	cpsr	cpsr	cpsr	cpsr	cpsr	cpsr	cpsr
	spsr_hyp	spsr_svc	spsr_abt	spsr_und	spsr_mon	spsr_irq	spsr_fiq

User mode and System mode share all their registers and r8~r14 registers are their own registers. They don't share those registers with other modes. For example, if processor mode changes from user mode to supervisor mode, the stack pointer changes from sp\_usr to sp\_svc. That means svc mode loads its own stack when activated. spsr register stands for Saved Program Status Register and when mode changes, the cpsr register of previous mode is saved to spsr register. Since user mode is not a privileged mode, and it can't access to cpsr, it doesn't have a spsr register.

All these core registers are also banked depending on its security mode. The security mode and non-security mode has its own register copies.

### 2.2.6 Security States

ARMv7-A defines the Secure state and Non-secure state. The Secure state is also known as Secure world while the Non-secure state is also known as Normal world.

In ARMv7-A, the introduction of the TrustZone Security Extension creates two security states for all processor modes, except Mon mode and Hyp mode, as shown in below table.

Processor Modes	Security States
User(USR)	Secure or Non-secure
FIQ	Secure or Non-secure
IRQ	Secure or Non-secure
Supervisor(SVC)	Secure or Non-secure
Monitor(MON)	Secure only
Abort(ABT)	Secure or Non-secure
Hypervisor(HYP)	Non-secure only
Undef(UND)	Secure or Non-secure
System(SYS)	Secure or Non-secure

### 2.2.7 Privilege Levels

The ARMv7 architecture defines different levels of *execution privilege*:

- in Secure state, the privilege levels are PL1 and PL0
- in Non-secure state, the privilege levels are PL2, PL1, and PL0.

PL0 indicates unprivileged execution in the current security state.

The current processor mode described in section 3.3.1 determines the execution privilege level, and therefore the execution privilege level can be described as the *processor privilege* level.

Every memory access has an *access privilege*, that is either unprivileged or privileged. The characteristics of the privilege levels are:

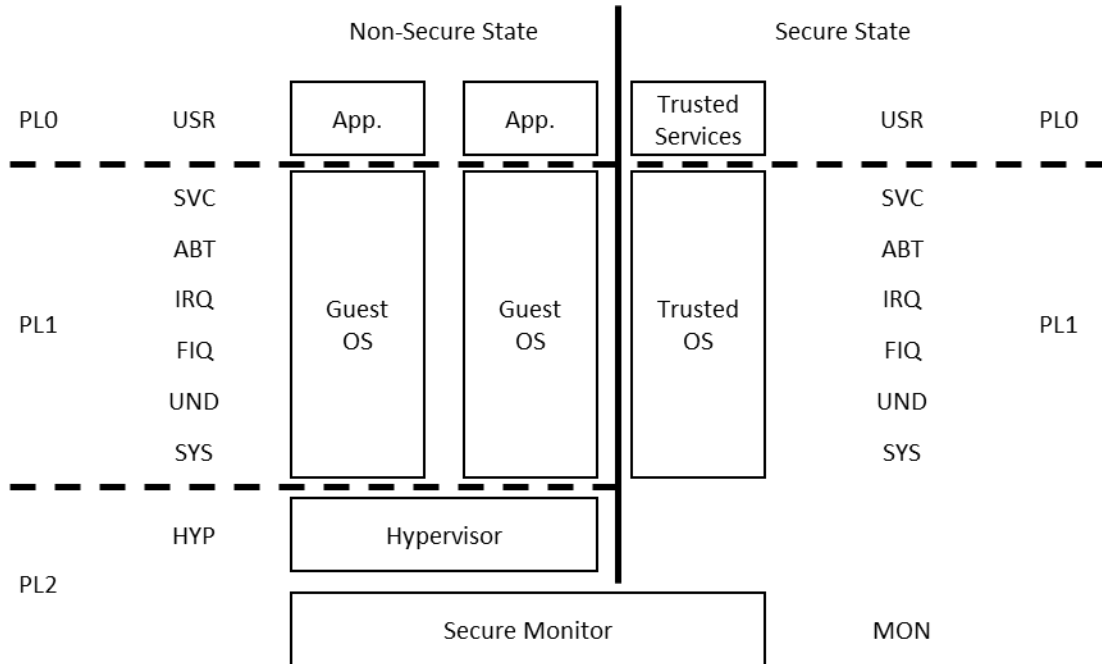
- PL0: The privilege level of application software, that executes in User mode. Therefore, software executed in User mode is described as unprivileged software. This software cannot access some features of the architecture. In particular, it cannot change many of the configuration settings.  
Software executing at PL0 makes only unprivileged memory accesses.
- PL1: Software execution in all modes other than User mode and Hyp mode is at PL1. Normally, operating system software executes at PL1. Software executing at PL1 can access all features of the architecture, and can change the configuration settings for those features, except for some features added by the Virtualization Extensions that are only accessible at PL2.  
The PL1 privilege refers to all the modes other than User mode and Hyp mode.  
Software executing at PL1 makes privileged memory accesses by default, but can also make unprivileged accesses.
- PL2: Software executing in Hyp mode executes at PL2. Software executing at PL2 can perform all of the operations accessible at PL1, and can access some additional functionality. Hyp mode is normally used by a hypervisor, that controls, and can switch between, Guest OSs, that execute at PL1. Hyp mode is implemented only as part of the Virtualization Extensions, and only in Non-secure state. This means that:

- 1) implementations that do not include the Virtualization Extensions have only two privilege levels, PL0 and PL1
- 2) execution in Secure state has only two privilege levels, PL0 and PL1.

In an implementation that includes the Security Extensions, the execution privilege levels are defined independently in each security state, and there is no relationship between the Secure and Non-secure privilege levels.

### **2.2.8 Relationship among Mode, Privilege, and Security**

Below figure?? shows how in the Non-secure state, you have PL0 for User mode, PL1 for exception modes, and PL2 for the Hypervisor, while in the Secure state, you have only PL0 and PL1, with Mon mode at PL1



Secure Monitor can access Secure and Non-secure state. Otherwise, code executing in either Secure or Non-secure state cannot use system registers associated with the other security state. For example, a Hypervisor executing at PL2 cannot access any of the Secure system registers.

### 2.2.9 Cortex-A9 and Cortex-A9 MPCore

Cortex-A9 processor implements the ARMv7 architecture and runs 32-bit ARM instructions, 16-bit and 32-bit Thumb instructions, and 8-bit Java™ bytecodes in Jazelle state.

Cortex-A9 MPCore is composed of one to four Cortex-A9 processors and *Snoop Control Unit(SCU)*. Snoop Control Unit is used to ensure cache coherency among processors. Cortex-A9 MPCore also has a set of peripherals such as global timer, watchdog, private timer, *Generic Interrupt Controller(GIC)*. Chip makers compose their system along with other IPs. The processor we will use in operating system implementation is Cortex-A9 MPCore.

## 2.3 Zynq-7000 Introduction

This chapter will describe an overview of zynq7000 and zc702 evaluation kit which we will develop our own operating system.

Zynq7000 SoC family is a good processors to practice in developing a custom operating system. Zynq7000 processor is developed by Xilinx which has dual or single-core ARM Cortex-A9 MPCore™ based *processing system(PS)* and Xilinx *programmable logic(PL)* in a single chipset. The ARM Cortex-A9 MPCore is a 32-bit processor core licensed by ARM Holdings implementing the ARMv7-A architecture [1]. Processing system has its own hardware resources such as I/O peripherals, memory interfaces, and can boot up without the help of programmable logic subsystem. User-programmed bitstream for programmable logic is downloaded during booting. Xilinx supports prebuilt IPs in their repository and

customer can use much of them for free. By using custom hardware programmed in a programmable logic subsystem, customer can achieve a high speed in processing of large volume of data. For example, bit manipulation on memory data(i.e. image rotation) performed by custom DMA module in PL is much faster than when it is done in PS. PS is used for generic operations like programming the PL subsystem, or high level applications like TCP/IP applications. So, customized hardware is used for high performance, repetitive programming and software running in PS is for flexible, high featured programming. This is called hardware-software codesign. Zynq7000 is well suited for this purpose. Now that we are going to design our own hardware and develop its device driver in chapter ???, zynq7000 well fits for this purpose.

The communication between PS and PL is done through the AXI4 interface. AXI4 interface is developed by ARM as a part of ARM *AMBA microcontroller bus architecture*. We don't need to know the details of AXI interface in operating system development but when we use xilinx vivado to design custom hardware, we have to touch a little bit of the AXI interface. You can get a detail specification of AXI interface from arm infocenter or refer xilinx document 0.

Figure 1.1 illustrates the top level picture of zynq7000 functional blocks [1]. The Zynq has following major functional blocks.

- Processing System
  - Application Processing Unit(APU)
  - Memory Interfaces
  - I/O peripherals(IOP)
  - Interconnect
- Programmable Logic(PL)

Xilinx composes the PS system by integrating each sub-blocks from different vendors. For example, the Cortex-A9 MPCore comes from ARM, USB is supplied from Synopsys, Gigabit Ethernet MAC from Cadence, etc. PL is derived by xilinx Artix-7 or Kintex-7 FPGA.

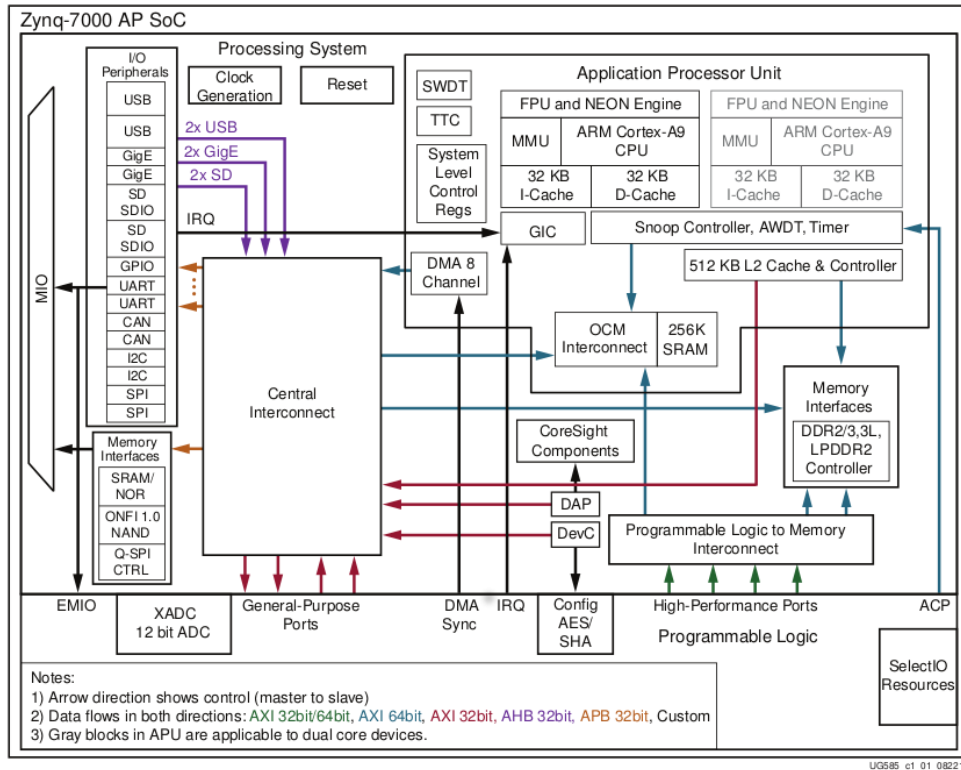


Figure 2.2.9-1 from xilinx UG585

## 2.4 Development PC Setup

### 2.4.1 Ubuntu 16.04 Installation

Ubuntu 16.04 with GCC tool chain are used for development PC. If you are using Windows PC, you first need to install Virtualbox and install Ubuntu 16.04 into the Virtualbox. I assume Windows 10 for host OS and Ubuntu 16.04 for guest OS. The Virtualbox can give the guest OS(Ubuntu 16.04) the abilities to connect to remote device through usb, uart serial connection, and abilities to access host OS(Windows 10)'s hardware resource. Installation Ubuntu and Virtualbox are not addressed in this book. You can refer [3], and [4] for download and installation. You can also get a lot of informations by googling.

### 2.4.2 Git Installation

Next, you have to install git into your Ubuntu guest OS. Git is another masterpiece of Linus Torvalds for managing Linux source tree. It isn't only used in many open source projects, but Microsoft Visual Studio also supports it now. Let's install git in your Ubuntu simply by running "sudo apt-get install git gitk". Gitk is a GUI tool to traverse the changes graphically. There is also a git running in Windows OS and you can install git for Windows.

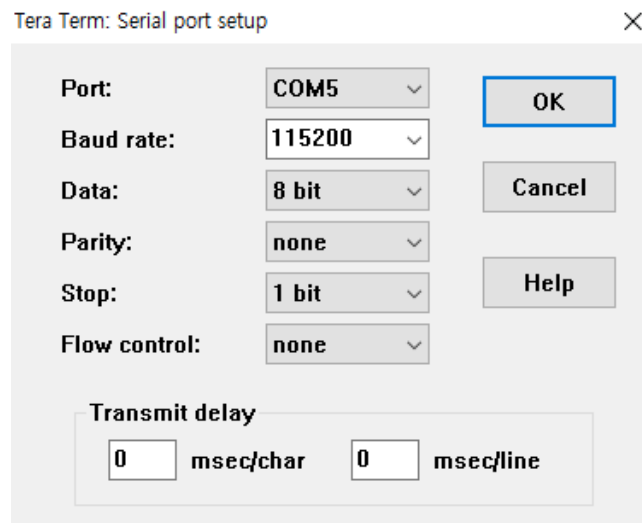
SLOS is managed by using git and all history and sources are uploaded to github.com. You can download SLOS source from github repository by running the command:

```
git clone https://github.com/chungae9ri/slos
```

I also recommend you git to manage your own sources and change history if you follow the steps described in this book. There are many good references on the internet.

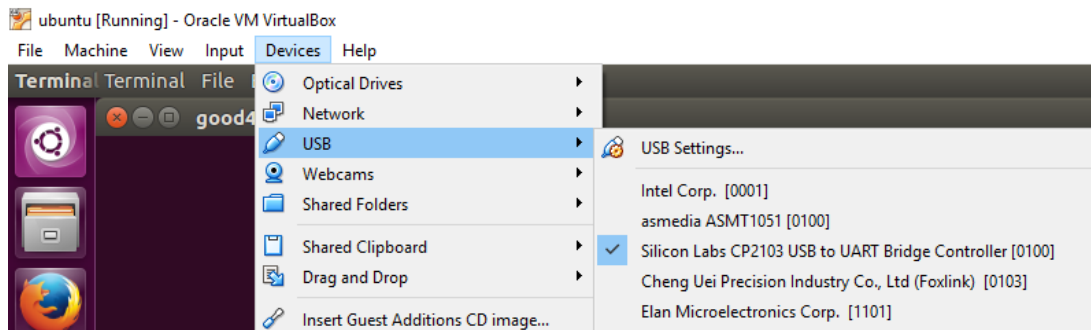
### 2.4.3 Serial Terminal Installation

After installing git, we have to install serial terminal to debug our operating system. Serial terminal is also used for shell task to receive user input. There are many serial communication applications and I recommend a teraterm. It is free and search teraterm on the internet, download it and install it. It is quite strait forward. Run teraterm program, choose Serial in new connection window, and press OK. After starting teraterm program, go to setup->serial port and set the serial port configuration as below. Port number(COM5 in below figure) is dependent on your system and you have to choose correct COM port number in your system.



**Figure 2.4.3-1 Teraterm serial port setting**

You might want to use a serial terminal in your Ubuntu guest OS. First you have to forward the serial port to your guest OS. For this, in your guest OS window, select Devices->USB and choose a serial to usb bridge as below.



**Figure 2.4.3-2 : USB to UART forwarding to guest OS**

Minicom application can be used for UART serial communication with target device and you can install it simply by “`sudo apt-get install minicom`” in your Ubuntu guest OS. Then start minicom with “`sudo minicom -s`”. Choose “Serial Device” by press shift+a and set Hardware Flow Control as “No” by pressing shift + f. Settings of minicom terminal looks like below image.

```

+-----+
| A -   Serial Device       : /dev/ttyUSB0 |
| B - Lockfile Location    : /var/lock     |
| C - Callin Program       :               |
| D - Callout Program      :               |
| E - Bps/Par/Bits         : 115200 8N1    |
| F - Hardware Flow Control : No           |
| G - Software Flow Control : No          |
|                                     |
| Change which setting? █               |
+-----+
  
```

**Figure 2.4.3-3 : Serial port settings**

The Serial Device could be different and is dependent on your PC. You can run “`dmesg | grep ttyUSB`” and find correct serial device(ttyUSBx) with a string “cp210x converter”. You can save the setting and run “`sudo minicom`” without setting option next time. You must see the booting messages when you turn on the Zynq evaluation board in step 2.4.



```

Starting kernel ...

Booting Linux on physical CPU 0x0
Linux version 4.4.0-xilinx (good4u@ohasi) (gcc version 4.9.2 20140904 (prerelease) (crosstool-NG linaro-1.13.1-4.9-2014.09 - Linaro GCC 4.9-2014.09) ) #2 SMP PREEMPT Tue Apr 11 16:52:50 EDT 2017
CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7), cr=18c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine model: axi_dbg
bootconsole [earlycon0] enabled
cma: Reserved 16 MiB at 0x1f000000
Memory policy: Data cache writealloc
PERCPU: Embedded 12 pages/cpu @debcd000 s19264 r8192 d21696 u49152
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 130048
Kernel command line: console=ttyPS0,115200 earlyprintk mem=512M memmap=512M$512M
PID hash table entries: 2048 (order: 1, 8192 bytes)
Dentry cache hash table entries: 65536 (order: 6, 262144 bytes)
Inode-cache hash table entries: 32768 (order: 5, 131072 bytes)
Memory: 493424K/524288K available (4840K kernel code, 218K rwdata, 1772K rodata, 2704K init, 213K bss, 14480K reserved, 16384K cma-reserved, 0K highmem)
Virtual kernel memory layout:
vector : 0xffff0000 - 0xffff1000 ( 4 kB)
fixmap : 0xffc00000 - 0xffff0000 (3072 kB)
vmalloc : 0xe0800000 - 0xff800000 (496 MB)
lowmem : 0xc0000000 - 0xe0000000 (512 MB)
pkmap : 0xbfc00000 - 0xc0000000 ( 2 MB)
modules : 0xbf000000 - 0xbfe00000 ( 14 MB)
.text : 0xc0008000 - 0xc067d364 (6613 kB)
.init : 0xc067e000 - 0xc0922000 (2704 kB)
.data : 0xc0922000 - 0xc0958820 ( 219 kB)
.bss : 0xc0958820 - 0xc098de7c ( 214 kB)

Preemptible hierarchical RCU implementation.
Build-time adjustment of leaf fanout to 32.
RCU restricting CPUs from NR_CPUS=4 to nr_cpu_ids=2.
RCU: Adjusting geometry for rcu_fanout_leaf=32, nr_cpu_ids=2
NR_IRQS:16 nr_irqs:16 16
slcr mapped to e0800000
L2C: platform modifies aux control register: 0x72360000 -> 0x72760000
L2C: DT/platform modifies aux control register: 0x72360000 -> 0x72760000
L2C-310 erratum 769419 enabled
L2C-310 enabling early BRESP for Cortex-A9
L2C-310 full line of zeros enabled for Cortex-A9
L2C-310 ID prefetch enabled, offset 1 lines
L2C-310 dynamic clock gating enabled, standby mode enabled
L2C-310 cache controller enabled, 8 ways, 512 kB
L2C-310: CACHE ID 0x410000c8, AUX_CTRL 0x76760001
zynq_clock_init: clk starts at e0800100
zynq clock init
sched_clock: 64 bits at 333MHz, resolution 3ns, wraps every 4398046511103ns
clocksource: arm_global_timer: mask: 0xffffffffffffffff max_cycles: 0x4ce07af025, max_idle_v

```

Figure 2.4.3-4 : Linux boot message in Zynq evaluation kit

## 2.4.4 GCC Tool Chain Installation

Now you have ubuntu desktop working, next thing for setting up your PC is installation of cross compiler. Keep in mind that our operating system will be running on ARM processor of PS in zynq7000 evaluation kit. Since the architecture of building machine and running machine is different, we need a cross compiler which compiles the sources according to target architecture. There are some commercial tool chains such as Mentor Graphics' Sourcery CodeBench, ARM's RVDS, but we will use GCC(GNU Compiler Collection) tool chain. You can download the GCC tools from ARM download site [9]. Currently custom operating system is built by 2017q1 GCC version. You can download it by running "git clone <https://github.com/chungae9ri/tools>". You can see the GCC file named "gcc-arm-none-eabi-6-2017-q1-update-linux.tar.bz2". Decompress it with "tar xvfj gcc-arm-none-eabi-6-2017-q1-update-linux.tar.bz2" and move it to ~/bin/ and rename it as "arm-2017q1". Add the path to tool chain executables to \$PATH environment variable. Open ~/.bashrc and add the path to the PATH environment variable like below.

```
export PATH=$HOME/arm-2017q1/bin:$PATH
```

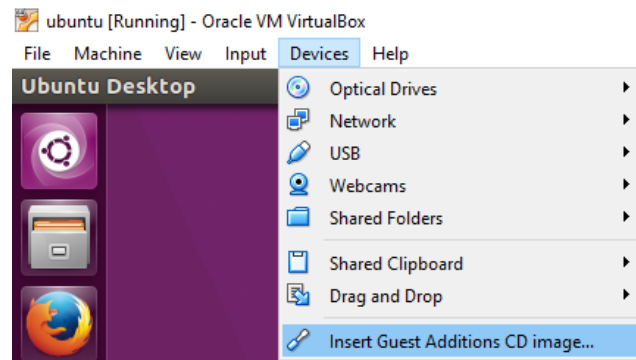
This also can be done in the Makefile to set the path to the toolchain. Adding the toolchain path to the Makefile will be covered in chapter 3.7. But I prefer setting the path in bashrc because I can use the GNU tools command anywhere in the shell by doing this.

## 2.4.5 Miscellaneous Settings

If finishing installation of ubuntu, let's do some fancy and convenient settings. Ubuntu needs "sudo" keywords for running a command with superuser authority. For example, if you want to edit a passwd file which needs a superuser permission, you must run "sudo vi /etc/passwd". If "sudo" command is run, it always asks a root passwd and it is very inconvenient to us, I think. Let's remove this annoying step. First run "sudo visudo" and add "Your account ALL=(ALL) NOPASSWD: ALL" into the latest line then save and exit. Now you can use "sudo" command without entering a passwd everytime.

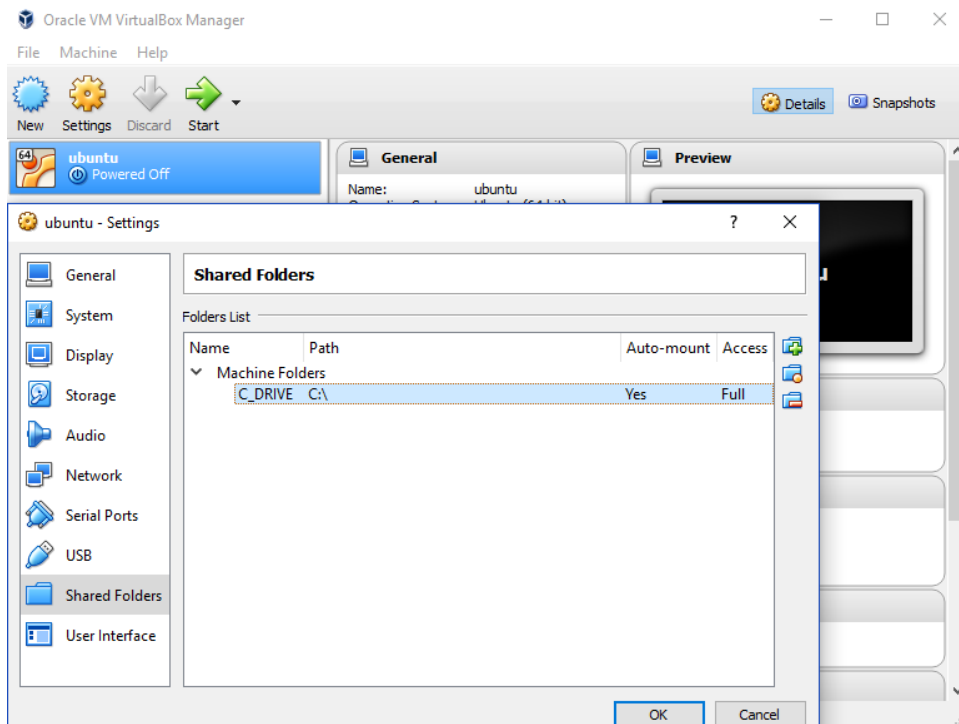
Next, change the default shell from dash to bash. After installing Ubuntu, the default shell is dash which is linked as /bin/sh. You can check this by running "ls /bin/sh -l" command. It will point to dash. Our development environment needs bash as default shell. Then run "sudo dpkg-reconfigure dash", and choose "NO" in the question screen.

After finishing Ubuntu16.04 installation, you have to install guest addition. Select Guest Addition image as Figure 2.3.1-1, enter passwd, then guest addition will be installed.



**Figure 2.4.5-1 : Run guest addition in Ubuntu guest OS**

Next, let's set a folder to share files between host OS and guest OS. Select the Shared Folders in Virtualbox's Settings window. Choose the whole directory, for example C: drive, or D: drive for shared folder and check auto mount. Then boot the Ubuntu guest OS and you can see the shared folder in /media/sf\_C\_DRIVE. Add your account to vboxsf group by running "sudo adduser \$your\_id vboxsf". This can give you a permission to access the shared folder. Now, you need to make a symbolic link to the shared folder. Run "ln -s /media/sf\_C\_DRIVE ~/c". You can access the shared folder by "cd c". This is very convenient when you work in Virtualbox guest OS. If you have another disk like D drive, you can also link this drive with "~/d" symbolic link. We'll make our development code in these driver and access guest OS with using these links.



**Figure 2.4.5-2 : Setting shared folder between Host OS and Guest OS**

## 2.5 Development Board Setup

Xilinx has a couple of evaluation kits and I assume we use ZC702 Evaluation Board for developing our own operating system. This board has dual core ARM Cortex-A9 application processor, 1G DDR3 memory, USB, Ethernet PHY, USB-to-UART bridge, status LEDs, User I/O for PS subsystem and Xilinx Artix-7 FPGA for PL subsystem. You can refer [2], [5] for detailed features. It seems to have over specification to develop a simple operating system, but it has also a well-defined development, debugging which is essential part for custom development.

For setting up the board, all we need to know is how to set the boot mode. Zynq-7000 can boot up through flash devices(master boot mode) which has following boot devices [2]

- Quad-SPI with optional Execute-in-Place mod,
- SD Memory Card,
- NAND,
- NOR with optional Execute-in-Place mode,

and for slave boot mode, zynq-7000 has JTAG boot.

We will use SD Memory Card for booting zynq-7000 and need to set the SW16 dip switch for SD card boot mode. Following is SW16 configurations for each options.

Boot Mode	SW16.1	SW16.2	SW16.3	SW16.4	SW16.5
JTAG mode	0	0	0	0	0

Independent JTAG mode	1	0	0	0	0
Quad SPI mode	0	0	0	1	0
SD mode	0	0	1	1	0
MIO configuration pin	MIO2	MIO3	MIO4	MIO5	MIO6

MIO configuration pin means boot mode pin which is connected to SW16. Setting the SW16 can program the system level boot mode register which stores the boot mode setting. This value is read while booting up and processed accordingly.

After setting the boot mode as SD mode, let's try to boot up the board. If you cloned the tools repository in step 2.3.2, you can find the bootloader and operating system images in the tools directory. The BOOT.BIN is for zynq7000 bootloader image which includes fsbl.elf(First Stage BootLoader), FPGA bitstream, and uboot.elf for loading Linux operating system. Image.ub is for Linux and ramdisk of root file system. You need to copy BOOT.BIN and image.ub files to micro SD card and insert it to Zynq evaluation board by using SD card adapter. Now after power-on, you can see the booting messages in your serial terminal (teraterm or minicom) if you correctly setup the serial terminal in chapter 2.3.3.

## 2.6 Xilinx SDK, Vivado, and Petalinux Installation

The PS subsystem itself is a standalone system and Xilinx provides bootloader and Linux binary image for high level operating system running on PS's ARM processor. To build custom operating system correctly running on the zynq chipset, we are going to Xilinx SDK to build final image. For this purpose, we will install Xilinx SDK including Petalinux, Vivado, and SDK.

### 2.6.1 Petalinux Installation

For installing Petalinux correctly, the Ubuntu PC must have additional packages installed. You can do this by running

```
"sudo apt-get install tofrodos gawk xvfb libncurses5-dev tftpd zlib1g-dev libssl-dev flex bison wget chrpath socat autoconf libtool texinfo libstd1.2-dev gcc-multilib zlib1g:i386"
```

After installing those packages, download Petalinux 2016.4 installer from [6]. Then install petalinux by running `"/petalinux-v2016.4-final-installer1.run ~/your_workspace/petalinux-v2016.4"`.

Petalinux is needed for building final BOOT.BIN which includes bootloader and our custom OS. We will use tools in Petalinux SDK in chapter ???.

### 2.6.2 Vivado Installatioin, Default Block Design, and Xilinx SDK

Vivado and SDK are necessary because it makes a basic bootloader and supply useful tools like XMD debugger. In addition, we can design our custom hardware and develop its device driver. We will define

our own hardware by using Verilog in Vivado IDE. The process for custom hardware design and device driver development are described in chapter ???.

Download the Vivado HLx 2016.4:Webpack Edition from [8]. Webpack edition is free and you can install free license.

## References

- [1] Xilinx User Guide : UG761 AXI Reference Guide
- [2] Xilinx User Guide : UG585 Zynq-7000 All Programmable SoC Technical Reference Manual
- [3] Xilinx User Guide : UG850 ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC
- [4] Virtualbox Download/Installation : <https://www.virtualbox.org/wiki/Downloads>
- [5] Ubuntu 16.04 Download : <https://www.ubuntu.com/download/desktop>
- [6] ZC702 Evaluation Kit Getting Started : <http://www.wiki.xilinx.com/Kits+ZC702+Getting+Started>  
[https://en.wikipedia.org/wiki/ARM\\_Cortex-A9](https://en.wikipedia.org/wiki/ARM_Cortex-A9)
- [7] Petalinux Download:
- [8] <https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools.html>
- [9] Vivado SDK Download : <https://www.xilinx.com/support/download.html>
- [10] GNU Cross Compiler : <https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads>
- [11]

### 3 Building Sources and SLOS Boot Up in Zynq7000

After finishing the development environment, next step is to bring up the board with a least set of our operating system and to print the famous “hello world” message to serial terminal. This base, skeleton code is composed of a linker script, reset vector handler, Xilinx simple UART driver, and main function for kernel entry point. Those are all we need for chapter 3.

I am going to describe the methods to develop custom operating system from this chapter. Download the source from the github repository.

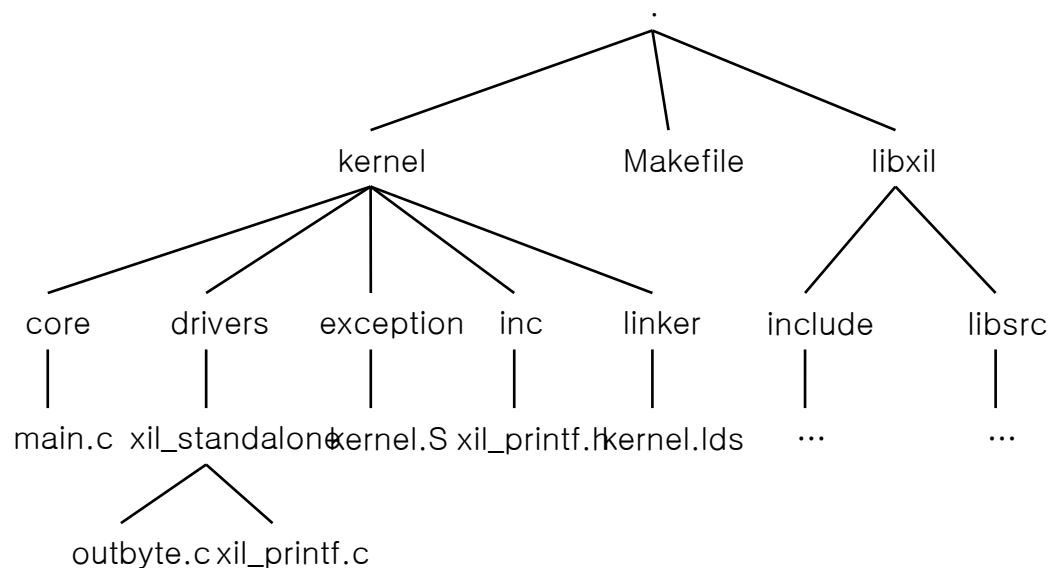
```
git clone https://github.com/chungae9ri/slos
```

```
git checkout -b your_branch_name chapter3
```

This source is only for your reference. It is the best for you to develop your custom operating system while going through this book.

#### 3.1 Base SLOS Sources

Let's first make an initial source tree for our own operating system. Figure 3-1 shows the initial stage of the source tree.



**Figure 2.6.2-1 : SLOS base source tree**

Kernel is a top level directory which has core, drivers, exception, inc and linker directories.

- **core** : Core directory has source files for process management, memory management, storage management, and other miscellaneous sources for implementing kernel core abilities.

Currently, core directory contains only main.c and other kernel core files will be added under this directory.

- drivers : Drivers directory has base bsp files such as UART. Currently, this directory has UART device driver only. This is used to print message to serial console. At the end of this chapter, we will print “hello world” message.
- exceptions : Directory for exception vectors and fault handlers. Currently, only reset vector handler which simply jumps to main entry is all for exception vector.
- inc : include directory for header files. Currently, header file for Xilinx UART device driver is here.
- linker : linker script directory.

libxil directory has xilinx’s bsp directory.

- libxil : include and source files for Xilinx UART device driver. It has many subdirectories for driving Xilinx hardware. These files are directly coming from Xilinx Zynq BSP.
- Makefile : Makefile for building source tree.

Currently, only basic sources to bring up Zynq7000 PS system will be added. Hardware initialization is done in fsbl bootloader and we don’t need to care about PS hardware initialization. This is very important in custom operating system development because hardware initialization itself is not the area of operating system development, but very painful to implement, and must be done before delving into OS development. If you are using other development board, you should find out the way to initialize your hardware. Mostly, bootloader does this and hardware initialization can be solved just by reusing vendor’s bootloader and correctly interfacing the bootloader with your operating system.

For your convenience, let’s make a directory named ‘slos’ to your C driver or D drive. Then you can access it from your guest Ubuntu OS by simply “cd ~/c/slos”. We can build our code from Ubuntu OS and debug it with host OS using Xilinx SDK tools.

### 3.2 Linker Script

Linker Script is used for linker to link input object files. Compiler compiles source codes and generate object files. Then, linker links these object files by solving function/variable references and generates final object file. While doing this, linker’s main roles are

- Merging input sections from input files to output sections
- Mapping memory layout of output sections

So, linker script describes those things for linker. Linker can do more but the all things we need to use in this book is those two things; section mapping and memory layout control. We don’t need to know more and after complete the first linker script, we will change a little. Let’s find out what is linker script and check out how it can be used in this operating system development. Most of the below sections comes from [1], and [2].



### 3.2.1 Linker Script Basics

The linker combines input objects into a single output object called as executable. Each object file has sections. A section in the input object is input section and a section in the output object is an output section.

Each section in an object file has a name and a size. Most sections also have an associated block of data, known as the section contents. A section may be marked as loadable, which means that the contents should be loaded into memory when the output file is run. A section with no contents may be allocatable, which means that an area in memory should be set aside, but nothing in particular should be loaded there (in some cases this memory must be zeroed out). A section which is neither loadable nor allocatable typically contains some sort of debugging information.

Every loadable or allocatable output section has two addresses. The first is the VMA, or virtual memory address. This is the address the section will have when the output file is run. The second is the LMA, or load memory address. This is the address at which the section will be loaded. In most cases the two addresses will be the same.

You can see the sections in an object file by using the `arm-none-eabi-objdump` program with the `‘-h’` option in your Ubuntu guest OS.

Every object file also has a list of symbols, known as the symbol table. A symbol may be defined or undefined. Each symbol has a name, and each defined symbol has an address, among other information. If you compile a C or C++ program into an object file, you will get a defined symbol for every defined function and global or static variable. Every undefined function or global variable which is referenced in the input file will become an undefined symbol.

You can see the symbols in an object file by using the `nm` program, or by using the `arm-none-eabi-objdump` program with the `‘-t’` option.

Linker scripts are text files. You write a linker script as a series of commands. Each command is either a keyword, possibly followed by arguments, or an assignment to a symbol. You may separate commands using semicolons. Whitespace is generally ignored.

Strings such as file or format names can normally be entered directly. You may include comments in linker scripts just as in C, delimited by ``/*’` and ``*/’`. As in C, comments are syntactically equivalent to whitespace.

### 3.2.2 Simple Linker Script

Below is the simplest linker script for SLOS running in Zynq7000 evaluation kit. This simple linker script is good enough to run our “Hello World” operating system. Let’s call this base linker script as “Hello World” linker script. We don’t need to change much from below base linker script and the basic structure of this linker script remains through this book. Add below linker script code to `kernel/linker/kernel.ld` and let’s look into the things what we need to know for our operating system development.

```
1  OUTPUT_ARCH(arm)
2  ENTRY(exceptions)
3
```

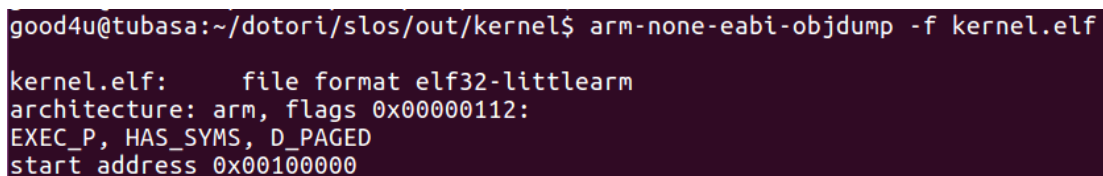
```

4  /* Define Memories in the system */
5  MEMORY
6  {
7      ps7_dds_0_S_AXI_BASEADDR : ORIGIN = 0x100000, LENGTH = 0x3FF00000
8  }
9
10 SECTION
11 {
12     . = 0x100000;
13     .text : {
14         *(EXCEPTIONS);
15         *(.text)
16     } > ps7_dds_0_S_AXI_BASEADDR
17     .data : {
18         *(.data);
19     } > ps7_dds_0_S_AXI_BASEADDR
20     .bss : {
21         *(.bss);
22     } > ps7_dds_0_S_AXI_BASEADDR
23 }

```

- OUTPUT\_ARCH(arm)

Specify a particular output machine architecture. The argument is one of the names used by the BFD library[3]. You can see the architecture of an object file by using the arm-none-eabi-objdump program with the '-f' option. You can see the architecture name "arm" in below screen capture. This command is optional/\* really? need to check \*/.



```

good4u@tubasa:~/dotori/slos/out/kernel$ arm-none-eabi-objdump -f kernel.elf
kernel.elf:      file format elf32-littlearm
architecture: arm, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00100000

```

- ENTRY(exceptions)

ENTRY command is used for setting entry point of program execution. Entry point is the first instruction of the program. The argument "exceptions" is a symbol which is defined in kernel.S. SLOS defines "exceptions" symbol as the start address of exception vectors which is 0x100000.

There are several ways to set the entry point. The linker will set the entry point by trying each of the following methods in order, and stopping when one of them succeeds.

- : the '-e' entry command-line option;
- : the ENTRY(symbol) command in a linker script;

: the value of the symbol start, if defined;  
: the address of the first byte of the '.text' section, if present;  
: The address 0.

- Both C and C++ type comments are allowed
- MEMORY Command

Although the linker can access the whole available memory, you can designate a memory region that the linker allocates to the sections. MEMORY command describes the location and size of memory region and it consists of region name, start address and length. It looks like below.

```
MEMORY
{
    name[attr] : ORIGIN = origin address, LENGTH = length
}
```

The name field is the name that linker refers to the region. This region name has no meaning outside of the linker script. Each region has different name.

The attr is an optional list to specify the attributes of particular memory region. The attr must consist of the following characters.

- R: Read-only section
- W: Read/write section
- X: Executable section
- A: Allocatable section
- I: Initialized section
- L: Same as I
- !: Invert the sense of any of the preceding attributes

ORIGIN is the start address of a specific memory region and LENGTH is its length. After a memory region is defined, the linker can place specific output sections into that memory region by using the >region in output section attribute. In the above Hello SLOS linker script, there is only one memory region defined as DDR memory region and its ORIGIN and LENGTH comes from the system-level address map in Zynq TRM document[1]. MEMORY command is optional, and it can be left out from the Hello SLOS linker script.

#### ▪ SECTIONS Command

SECTIONS command is the most important command in linker script. It does the basic things that the linker script must do: It maps input sections to output sections and place the memory layout of output sections. The SECTIONS command has a series of symbol assignments and output section descriptions enclosed in curly braces. The first line inside the SECTIONS command of the Hello SLOS linker script sets the value of the special symbol '.', which is the location counter. If you do not specify the address of an output section explicitly, the address is set from the current value of the location counter. The location counter is then incremented by the size of the output section. At the start of the 'SECTIONS' command, the location counter has the value '0'. Line 12 in example linker script sets current location counter as 0x100000. After that, the location counter is increased by the amount of output section size or you can set it with a specific number.

Every loadable or allocable output section has below two addresses.

- VMA(Virtual Memory Address): This is the address the section will have when the output binary run.
- LMA(Load Memory Address): This is the address at which the section will be loaded.

In most cases the two addresses will be the same, we always use both addresses are same.

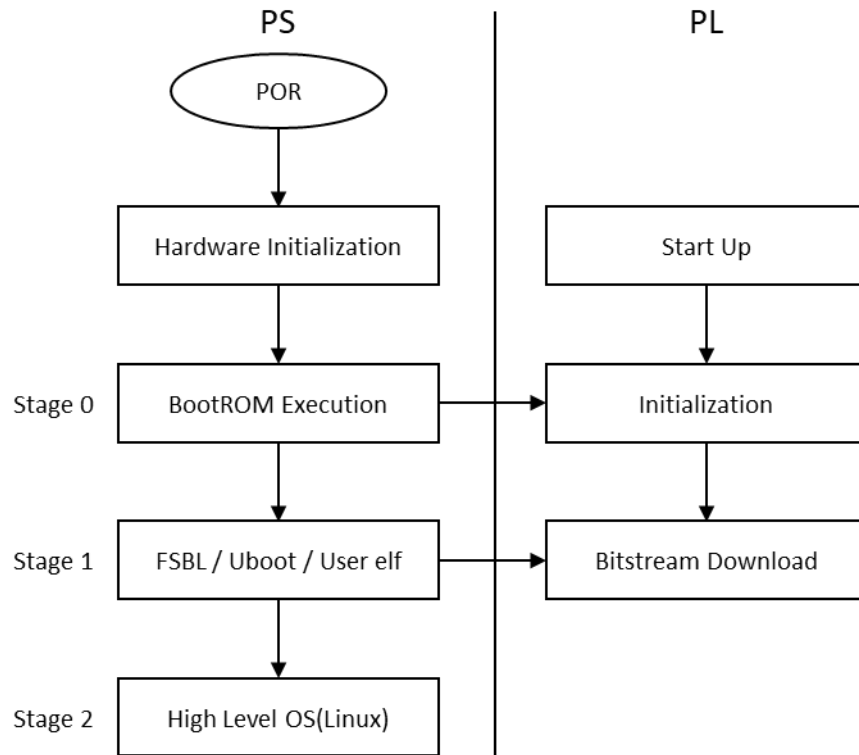
.text in line 13 designate the text output section. Text section is composed of EXCEPTIONS input sections and .text input sections from all input files. The wild card '\*' is used for any file names. That is, \*(.text) means the .text sections of all input files, and these input sections go to the .text output section which starts from current location counter starting from 0x100000. This SECTION command does the mapping from input sections to output section and memory layout of output sections.

.data section and .bss section are done simply by the same way with .text section. All .data sections of input files go to .data output section. All .bss input sections for uninitialized data go to .bss output section.

After you successfully build the Hello SLOS kernel source, you can see the sections of kernel.elf by running "arm-none-eabi-objdump -h kernel.elf" in out/kernel directory. You can see there are many other sections which are mapped to output sections without using linker script. You can refer [??] for more details on linker script, but the basic Hello SLOS linker script is enough for this simple operating system development. At the end of chapter 3, we can run Hello SLOS with this simple linker script.

### 3.3 Zynq7000 booting sequence

Below Figure ?? summarizes the Zynq7000 booting sequence. There are 3 booting stages we need to remember after POR(Power On Reset); BootROM Execution at Stage 1, FSBL / UBoot / User Code Execution at Stage2, and High Level OS Booting at Stage 3. Stage 2 is our primary concern. We will use Stage 2 in operating system development. Stage 2 booting binary is composed of 3 separate binaries; fsbl elf, PL bitstream, and UBoot elf or User built elf. At Stage 2, First, FSBL runs and it initialize registers, download FPGA bitstream to PL subsystem, does other important initializations, and jumps to UBoot or User built elf application. If FSBL jumps to UBoot, UBoot will finally load Linux OS and jumps to the start point. FSBL also can load User-built elf binary and jumps to it. This stand-alone executable can be used to verify custom hardware in PL subsystem. We will use this User-built binary for operating system development. We are going to build our operating system as a elf executable and attach it to the third part of Stage 2 binary. By following this way, we don't need to take care of hardware initializations. Those are done in FSBL and we can use a verified bootloader. Hardware initialization is non-operating system part and almost impossible to implement if we do that from scratch.



### 3.4 Exception Vector and Reset Handler

Processor has exception vector for asynchronous events. One example of those asynchronous events is interrupt. If such event occurs, the processor stops current execution and jumps to the predefined location which is called *exception vector*. ARM has Reset, Undefined Instruction, Supervisor Call, Prefetch Abort, Data Abort, Reserved (not used in this development), IRQ Interrupt, FIQ Interrupt exceptions. Since the handlers have only 4 bytes space, all they can do is jumping to the detailed implementation routine for handling each exception. Below Table summarized the exception vector used for the operating system development.

Exception	Offset from vector base	Mode on entry	CPSR.F bit on entry	CPSR.I bit on entry	Action
reset	0x00	supervisor	set	set	branch to its handler address
undefined instruction	0x04	undefined	unchanged	set	
supervisor call	0x08	supervisor	unchanged	set	
prefetch abort	0x0c	abort	unchanged	set	
data abort	0x10	abort	unchanged	set	
reserved	0x14	reserved	-	-	

irq	0x18	irq	unchanged	set	
fiq	0x1c	fiq	set	set	

The start address of exception handlers is set by *VBAR (Vector Base Address)* and each handler placed 4 bytes shifted from this base address. The VBAR is set by system control register, cp15. Since our 'Hello World' linker script sets the exception address as 0x100000, we have to set the VBAR value to the same value, 0x100000. The VBAR can be set as below

```
1    ldr    r0, =0x100000
2    mcr    p15, 0, r0, c12, c0,
```

The line 2 uses a special ARM instruction to set the system level control register. We are going to learn these system level registers (coprocessor registers) in ??, then currently, just setting the VBAR with these two lines is enough for the 'Hello world' operating system.

The "Hello World" operating system implements only reset vector and all other exceptions are not necessary. Let's add kernel.S file to the kernel/exception directory and add below base code for reset vector to kernel.S.

```
1    /* arm exception code */
2    .extern main
3    .section EXCEPTIONS, "ax"
4    .arm
5    .global exceptions
6    exceptions :
7        b reset_handler
8
9    reset_handler:
10        /* jump to main */
11        b      main
12    .end
```

Line 1 is for comment and line 2 is for specifying the extern symbol. In this case the kernel main routine function is the extern symbol. It is used for jump address at the end of reset vector. Line 3 is for specifying the section name of the routine. This section is mapped to the first location of input section in the "Hello World" linker script. Line 4 is for specifying ARM architecture and Line 5 is for specifying the global symbol(or address) for exception vector. This symbol is used for the entry point in "Hello World" linker script line 2. From this place, the entry routine of exception vector needs to be implemented and the entry routine must be reset vector which has offset 0 from the base address. As I mentioned, the reset vector does jump to other address (Line 7) which is the beginning of detailed implementation. The reset\_handler symbol is the beginning of reset vector implementation. The reset vector only jumps to kernel main entry address which is imported in line 2.

### 3.5 Kernel Main Entry Point

Kernel main entry point of this “Hello World” operating system prints a “hello world” message to console which is set in chapter ??? and falls into infinite idle loop. Kernel main entry looks like below.

```
1    #include <xil_printf.h>
2
3    void cpuidle()
4    {
5        /* do nothing for now */
6        for (;;)
7    }
8
9    int main(void)
10   {
11       xil_printf("###hello world\n");
12       cpuidle();
13
14       return 0;
15   }
```

Line 1 is for using Xilinx UART driver which will send messages to console terminal. How to include the UART driver sources will be explained in next chapter. Line 3 to line 7 is the cpu idle function which have the processor spin forever. The main entry point in line 9 does print “hello world” message through Xilinx xil\_printf() function which is running on top of Xilinx UART driver. Then the main function goes to infinite loop of cpuidle(). The main loop never returns.

### 3.6 Porting Xilinx UART Drivers

Before custom OS development, we need device drivers for basic hardware. Those are timer interrupt for process management and UART serial device driver for shell commands. Those drivers are hardware specific and developing those drivers is not our concern. We will design our own hardware and develop its device driver later. Since all “Hello World” operating system does is printing “hello world” message, it needs only UART driver for now. The sources for “Hello World” operating system downloaded from git repository already has UART device driver in libxil/libsrc, and kernel/drivers folder. So, if you download the base sources, you can use it without any concern of Xilinx UART. But, if you want to know how it works, let’s first port verified Xilinx UART device driver to our BSP.

To import the Xilinx UART driver, we first make their basic bsp libraries which has all device drivers for driving the hardware. If you correctly generate the default block design in chapter 2.6.2, you should see fsbl\_bsp in your eclipse project explorer. This Xilinx fsbl bsp is a good resource repository for us to import device driver sources of their hardware. Copy the fsbl\_bsp/ps7\_cortexa9\_0/libsrc/uartps\_x\_x to under libxil/libsrc. If we want other device drivers, then copy them also to the same path, but we only need a UART driver for printing “Hello World” message.

## 3.7 Makefile and SLOS Build Process

### 3.7.1 Overview of Makefile

*GNU Make* lets us define how to compile sources and how to link the object files and finally how to generate the final executable binary. Knowing GNU Make itself takes time, but we need a small portion of Make functionalities for our development and will cover those necessary parts only.

First, let's get basic idea on what is Make and how it works. GNU Make does its work in two distinct phases. During the first phase it reads all the makefiles, included makefiles, etc. and internalizes all the variables and their values, implicit and explicit rules, and constructs a dependency graph of all the targets and their prerequisites. During the second phase, make uses these internal structures to determine what targets will need to be rebuilt and to invoke the rules necessary to do so [???].

Understanding these two phases is important. For example, there are two types of variable assignment depending on the phases. If the assignment happens in the first phase, it is called *immediate assignment*. If the assignment is in second phase, it is *deferred assignment*. We use “:=” for immediate assignment and “=” for deferred assignment. Mostly, I am using immediate assignment.

*/\* describe target, dependency, and rule here \*/*

The basic makefile needs “rules” that define how to generate the output of makefile. It looks like this:

target ... : prerequisites ...

recipe

....

A *target* is usually one of the output names which is expected to be generated by this makefile. Executables or object files can be the target name. There are standard targets for makefile. “*all*” target is a necessary target for all makefiles. “*all*” target needs entire program to be compiled. This is a default target.

A *prerequisite* is the inputs to generate the target. Target is dependent on the prerequisites and if any changes in the prerequisites or any prerequisites are newer than target force make to rebuild the target.

A recipe is the definition of action how the make create target. Mostly this recipe uses the prerequisites for its input but sometimes it doesn't have prerequisites as inputs like “*clean*” target.

Below is a simple makefile to build “hello” executable.

```
1 all : hello
2 hello : main.o hello.o
3     gcc -o $@ $^ -lc
4 main.o : main.c
5     gcc -c $^
6 hello.o : hello.c
7     gcc -c $<
8 clean :
9     rm *.o hello
```



The make first search the first target which is “all” target in line 1, then it will find the “all” target depends on “hello”. Now make create the dependency graph for all these targets through line 2 to line 7. Then make will generate those targets with bottom-up traversing the graph.

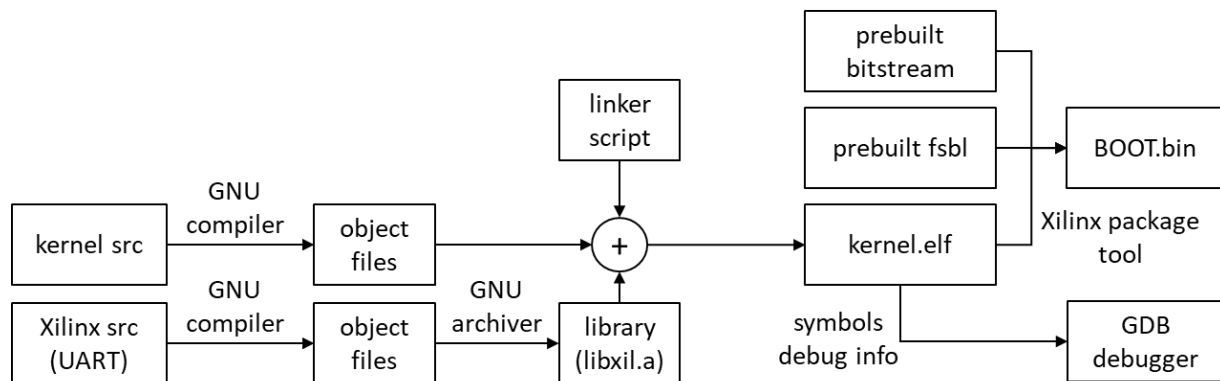
The value of variable is referenced by using “\$” symbol in makefile but in this simple makefile, the \$ mark is used other way for “Automatic Variables”. The document ??? explains more automatic variables.

- \$@ : The file name of target of the rule.
- \$^ : The names of all prerequisite files with white space between them.
- \$< : The name of first prerequisites.

The line 3, “-lc”, is used for linking a library whose is “libc.a”. Be careful the library name which is used only after “lib” and the extension is omitted. If the library name is “libhello.a”, then the syntax for linking this library must be “-lhello”. Option “-L” defines the path of library.

### 3.7.2 SLOS build process

Below is the build process of “Hello World” operating system. We will add a ramdisk binary in chapter 6.???. Now we only have kernel process and the build process looks like below. The Makefile of “Hello World” operating system does the steps to build kernel.elf and Xilinx package tool merges prebuilt PL bitstream and prebuilt fsbl binary with kernel.elf to generate final BOOT.bin image.



Below Makefile is for the steps to generate the kernel.elf.

```

1  export PATH:=$(HOME)/bin/arm-2017q1/bin:$(PATH)
2
3  LIBS := $(HOME)/bin/arm-2017q1/arm-none-eabi/lib
4  LIBS2 :=$(HOME)/bin/arm-2017q1/lib/gcc/arm-none-eabi/6.3.1
5  CC := arm-none-eabi-gcc
6  ASM := arm-none-eabi-as
7  LD := arm-none-eabi-ld
8  AR := arm-none-eabi-ar
9  OBJCOPY :=arm-none-eabi-objcopy
10 LIBXIL := libxil.a
11
12 TOP_DIR :=$(shell pwd)
13 OUT_TOP := $(TOP_DIR)/out
14
15 KERNMODULES := core exception drivers/xil_standalone
16 KERNSRCDIR := $(addprefix kernel/,$(KERNMODULES))
17 KERNOUTDIR := $(addprefix out/kernel/,$(KERNMODULES))
18
19 KERNCSRC := $(foreach sdir,$(KERNSRCDIR),$(wildcard $(sdir)/*.c))
20 KERNCOBJ := $(patsubst %.c,out/%.o,$(KERNCSRC))
21 KERNASMSRC := $(foreach sdir,$(KERNSRCDIR),$(wildcard $(sdir)/*.S))
22 KERNASMOBJ := $(patsubst %.S,out/%.o,$(KERNASMSRC))
23
24 LIBMODULESTEMP:= uartps_v3_3
25 LIBMODULES:= $(addsuffix /src, $(LIBMODULESTEMP))
26 LIBSRCDIR := $(addprefix libxil/libsrc/,$(LIBMODULES))
27 LIBOUTDIR := $(addprefix out/libxil/libsrc/,$(LIBMODULES))
28 LIBCSRC := $(foreach sdir,$(LIBSRCDIR),$(wildcard $(sdir)/*.c))
29 LIBCOBJ := $(patsubst %.c,out/%.o,$(LIBCSRC))
30 LIBASMSRC := $(foreach sdir,$(LIBSRCDIR),$(wildcard $(sdir)/*.S))
31 LIBASMOBJ := $(patsubst %.S,out/%.o,$(LIBASMSRC))
32
33 INC := -I$(TOP_DIR)/kernel/inc -I$(TOP_DIR)/libxil/include
34 LDS :=$(TOP_DIR)/kernel/linker/kernel.lids
35
36 vpath %.c $(KERNSRCDIR)
37 vpath %.S $(KERNSRCDIR)
38 vpath %.c $(LIBSRCDIR)
39 vpath %.S $(LIBSRCDIR)
40
41 define make-obj
42 $1/%.o: %.c

```

```

43     $(CC) $(CFLAGS) $(INC) -o $$@ -c $$< -g -mcpu=cortex-a9 -mfpv=vfpv3 -mfloat-abi=softfp
44
45     $1/%.o: %.S
46         $(CC) $(INC) -o $$@ -c $$< -mcpu=cortex-a9 -mfpv=vfpv3 -mfloat-abi=softfp
47     endif
48
49     $(foreach bdir, $(KERNOUTDIR),$(eval $(call make-obj,$(bdir))))
50     $(foreach bdir, $(LIBOUTDIR),$(eval $(call make-obj,$(bdir))))
51
52     all: checkdirs $(LIBXIL) kernel.elf
53
54     checkdirs : $(LIBOUTDIR) $(KERNOUTDIR)
55
56     $(LIBOUTDIR) :
57         mkdir -p $$@
58
59     $(KERNOUTDIR) :
60         mkdir -p $$@
61
62     $(LIBXIL) : $(LIBCOBJ) $(LIBASMOBJ)
63         $(AR) rc $(OUT_TOP)/libxil/$@ $(LIBCOBJ) $(LIBASMOBJ)
64
65     kernel.elf : $(KERNCOBJ) $(KERNASMOBJ)
66         $(LD) -T $(LDS) -o $(OUT_TOP)/kernel/kernel.elf $(KERNCOBJ) $(KERNASMOBJ) -
67     L$(OUT_TOP)/libxil -L$(LIBS) -L$(LIBS2) -lxil -lc -lgcc
68
69     clean :
70         rm -rf $(OUT_TOP) libxil.a
71         rm -f libxil/*.a $(LIBCOBJ) $(LIBASMOBJ)

```

Line 1 is for setting the path to the GNU tool chain. This path also can be added to .bashrc file as described in chapter 2.4.4. If we use deferred assignment (=) for \$PATH environment variable instead of immediate assignment (:=), the make will fail to find the GNU tool chain binaries. So it is important to know the two phases of make process. Line 3~9 are for setting the GNU tool binaries, and path to the libraries. Line 15~17 defines the subdirectories of kernel source files and the output folder of compiled object files. Line 19~22 list up the source files which has .c or .S file extension. Line 24~31 does the same process for Xilinx BSP library. In this simple startup phase, it has only UART device driver. Line 33 is for defining the include path which is used in compiling the sources in line 43 and 46. Line 36~50 are the rules to build the object file from source files in all directories. Line 52 is the target of this makefile. It first creates the corresponding output directories under out folder. Line 62~63 is for generating the Xilinx BSP library by using GNU archiver (arm-none-eabi-ar). The archiver merges the BSP output object files to libxil.a file. Finally the last dependency of target, kernel.elf, is generated in line 65~67. Line 69 is another target

of this makefile which cleans the build by removing all the output objects. This makefile describes the process for creating kernel.elf and covers the path from kernel src and Xilinx src to kernel.elf in figure ???.

A function call in makefile is similar with variable reference. The syntax of function call looks like this:

`$(function argument)`

or like this:

`${function argument}`

*function* is function name. The functions used in this makefile is explained below. For detailed information, and for other functions supported in makefile, refer ??? document.

- `$(addprefix prefix, names ...)`  
This function add prefix to the series of names. names is a series of string separated by white space. For example, line 16 `$(addprefix kernel/, $(KERNMODULES))` output is kernel/core kernel/exception kernel/drivers/xil\_standalone.
- `$(wildcard pattern)`  
The argument *pattern* is a file name pattern, typically containing wildcard characters. The result of wildcard is a space-separated list of the names of existing files that match the pattern.
- `$(foreach var,list,text)`  
This function is used for repetitive iteration of *text* with *var* which is assigned values from *list*. Line 19, `$(foreach sdir,$(KERNSRC_DIR),$(wildcard $(sdir)/*.c))` means all .c files in each `$(KERNSRC_DIR)`. `$(KERNSRC_DIR)` variable is assigned as kernel/core kernel/exception kernel/drivers/xil\_standalone in line 16.
- `$(patsubst pattern,replacement,text)`  
This function is for pattern substitution. It replaces the *pattern* in *text* with *replacement*. Line 20, `$(patsubst %.c,out/%.o,$(KERNCSRC))` replaces the .c extension of kernel source files with out/src\_file\_name.o. '%' is wildcard character for matching any number of any characters within word.
- `$(eval ...)`  
The argument to the eval function is expanded, then the result of that function are parsed as makefile syntax. The expanded results can define new variables, targets, implicit or explicit rules.
- `$(call variable param1, param2,...)`  
This function can call user defined function with parameter *param1*, *param2*. param1 is referenced in the function as `$(1)` and param2 is referenced as `$(2)` and so on. The `$(0)` is for *variable*. The *call* function can be used with *eval* function to define new makefile rules. Line 49, 50 are for this purpose which define an explicit rules of target object files and their dependency.

### 3.7.3 Booting SLOS in The Development Board

## 3.8 Debugger in SLOS

### 3.8.1 How to debugging

Debugger is always important in development. We have to prepare a good debugger before going further. We are going to combine the Xilinx Microprocessor Debugger (XMD) and GNU debugger to debug our simple operating system. XMD is attached to ARM Cortex processor by using JTAG interface and export GDB interface to remote debugging. XMD has more features but we are going to use only XMD with GDB. XMD and GDB are all included into SDK installation. If you properly install Vivado and its SDK in chapter 2.6.2, you can use all SDK tools simply by running the settings.bat file. Open a command window in your host Windows OS and go to your SDK installation directory. You can see the settings.bat file, then run it. After turning on the power, you should see “hello world” message in your serial terminal. At this point the cpu is spinning forever. Now let’s connect XMD debugger. Run “xmd” command in your command window. Since the settings batch file sets up the environment variables, the xmd command properly runs from everywhere. In the XMD prompt, run “connect arm hw”. This command will connect the XMD debugger with ARM cortex-A9 processor with JTAG. You should see a message like below.

```
C:\Xilinx\SDK\2016.4>xmd
***** Xilinx Microprocessor Debugger (XMD) Engine
***** XMD v2016.4 (64-bit)
**** SW Build 1756540 on Mon Jan 23 19:11:23 MST 2017
** Copyright 1986-2016 Xilinx, Inc. All Rights Reserved.

WARNING: XMD has been deprecated and will be removed in future.
        XSDB replaces XMD and provides additional functionality.
        We recommend you switch to XSDB for commandline debugging.
        Please refer to SDK help for more details.

XMD%
XMD% connect arm hw

JTAG chain configuration
-----
Device   ID Code      IR Length  Part Name
  1      4ba00477         4      arm_dap
  2      23727093         6      xc7z020
-----

Enabling extended memory access checks for Zynq.
Writes to reserved memory are not permitted and reads return 0.
To disable this feature, run "debugconfig -memory_access_check disable".

-----

CortexA9 Processor Configuration
-----
Version.....0x00000003
User ID.....0x00000000
No of PC Breakpoints.....6
No of Addr/Data Watchpoints.....4

Connected to "arm" target. id = 64
Starting GDB server for "arm" target (id = 64) at TCP port no 1234
XMD%
```

Next step is connecting GDB debugger to XMD’s GDB remote server. While running XMD, it starts a GDB server on port 1234. So, let’s connect a GDB debugger to this and debug our simple hello world OS. Open another command window, go to the SDK installation directory and run settings.bat. Then go to the slos root directory and run “arm-none-eabi-gdb out/kernel/kernel.elf”. Now we can connect GDB server

by running “target remote localhost:1234” in GDB command prompt. If everything goes well, cpu stops at infinite spinning loop and you should see below message.

You can add repetitive GDB configurations into a specific file, for example a .gdbinit file, and automatically configure the arm-none-eabi-gdb by using this configuration file. If .gdbinit file is in the same path with the gdb binary, gdb will auto-configure with .gdbinit. If .gdbinit file is not in the same directory with arm-none-eabi-gdb binary, “-x path\_to\_gdbinitfile” option is needed. Open a file which text editor and add “target remote localhost:1234” to the file and save it as “gdbinit.txt”. When you run GDB, you need to designate the configuration file path like a below example.

```
arm-none-eabi-gdb out/kernel/kernel.elf -x c:\gdbinit.txt
```

Then the GDB will automatically connect to XMD gdb server.

```
C:\dotori\slos>arm-none-eabi-gdb out/kernel/kernel.elf
GNU gdb (Linaro GDB 2016.07) 7.11.1.20160714-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-w64-mingw32 --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from out/kernel/kernel.elf...done.
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
cputidle () at kernel/core/main.c:8
8      for (;;)
(gdb)
```

Now, we are ready to use our debugger. Let’s break at the reset vector. Reset vector address is 0x100000 which is defined in our basic linker script in chapter 3.2.2. Below is the step to break at reset vector.

- 1) From the gdb prompt screen in the upper figure, type ‘c’ which means continuing running the program.
- 2) Switch to XMD window and type “rst” which means reset the processor. Then the processor resets at 0x00000000. This also breaks the GDB at the same address.
- 3) In the XMD debugger window, type “bps 0x100000 hw”. This makes a hardware break point at address 0x100000 which is the address of reset vector. If software break point is set, the processor might miss this break point. Hardware break that uses processor register saving break address to be compared with PC (Program Counter) is stronger than software break that inserts a trap code into the break address.
- 4) Switch to the GDB debugger window and type ‘c’ to continue program.

Then, the processor stops running at the address 0x10000. You can see the source code, check values, and so on. For example, run the “disassem” command, then GDB displays the assembler code. You can go to next assembler instruction by running “steppi or si” command.

### 3.8.2 XMD and GDB Commands

A summarized list for frequently used XMD command and GDB command follows.

XMD commands

Commands	Description	Example Usage
bpl	Lists break points.	bpl
bpr	Removes break point(s).	bpr 0x400 bpr main bpr all
bps	Sets break point.	bps 0x400 bps main hw
con	Continues from current PC or optionally specified address.	con con 0x400
mrd	Reads memory location(s) starting at <address>. Options: mrd <address> [number of words   half words   bytes] {w h b}	mrd 0x400 mrd 0x400 10 mrd 0x400 10 h
mrd_var	Reads memory corresponding global variable in the loaded elf file.	mrd_var var1 kernel.elf
mwr	Writes memory location(s) starting at <address>. Options: mwr <address> <values> [number of words   half words   bytes] {w h b}	mwr 0x400 0x12345678 mwr 0x400 1 h mwr 0x400 {0x12345678 0x87654321} 2
rrd	Reads all registers or <num> register. Options: rrd [reg_num]	rrd rrd r1 or rrd 1
rst	Resets the system . The processor will stop at the processor reset location.	rst
rwr	Writes register <reg_num> with <hex_val> Options: rwr <reg_num reg_name> <hex_val>	rwr pc 0x400
srrd	Reads special purpose register or read <reg_name> register.	srrd srrd pc
state	Displays the current state of processor.	state
stop	Stops the target processor.	stop
watch	Sets a read or write watch point at <address> . If the values compares to <data>, stops the processor. Don't care values are specified by X. Options: watch {r w} <address> <data>	watch r 0x400 0x1234 watch r 0x40X 0x12X4

#### GDB Commands

Start / Stop commands		
r(un)	Start your program	run
q(uit)	Exit GDB	quit
Breakpoints		

b(reak)	Set break point a line, address, function, offset lines and if expression is true. Options: break [file:] {line function} break {+ -} offset break *addr break break ... if expr	break main.c:10 break main.c:main break +10 break *0x400 break main.c:37 if var==1
Execution control commands		
c	Continue running programing	c
s(step)	Executes another line reached or count times if specified. Options: step [count]	step step 100
stepi si	step by machine instructions rather than source lines. Options: stepi [count]	stepi stepi 100
n(ext)	Executes next line, including any function calls. Options: next [count]	next next 100
nexti ni	Next machine instruction rather than source lines. Options: nexti [count]	nexti nexti 100
until	Runs until next instruction or location. Options: until [location]	until until 100
finish	Runs until selected stack frame returns.	finish
info b(reak)	Display breakpoints.	info b
clear	Deletes breakpoint at next instruction, at function, and on source line. Options: clear [file:]{fun line}	clear clear main.c:main clear main.c:10
delete	Deletes breakpoints.	delete
enable	Enables breakpoints or breakpoint n. Options: enable [n]	enable enable 2
disable	Disables breakpoints or breakpoint n. Options: disable [n]	disable disable 2
ignore	Ignores breakpoint n, count times. Options: ignore n count	ignore 2 100
Stack information commands		
backtrace	Prints trace of all frames in stack or of n. Options: backtrace [n]	backtrace
frame	Selects frame number n or frame at address n; if no n, display current frame. Options: frame [n]	frame
info args	Arguments of frame	info args
info locals	Local variables of selected frame	info locals



info reg info all-reg	Displays register values in selected frame or all-reg includes floating point registers. Options: info reg [rn]	info reg info reg r1
Display commands		
p(rint)	Shows value of expr or last value of history according to format /format. /x: hexadecimal, /d: signed decimal, /u: unsigned decimal, /o: octal, /t: binary, /a: address, /c: character, /f: floating point. [expr] can be variable or address. Options: print [/x] [expr] print [/d] [expr] print [/u] [expr] print [/o] [expr] print [/t] [expr] print [/a] [expr] print [/c] [expr] print [/f] [expr]	p/x *0x100018 p/c cval p/d dval p/f fval
x	Examines memory at address expr according format /format. /N: count of how many units to display. /u: unit size; one of /b(byte), /h(half word), /w(word), /g(double word). /f: printing format; one of /s(null terminating string), /i(machine instruction). Options: x [/Nuf] expr	x 0x100000 x /20i 0x100000
disassem	Displays memory as machine instructions. Options: disassem [addr]	disassem 0x100000
Working / Source Files		
file	Uses file for both symbols and executables.	
symbol	Uses symbol table from file; or discard. Options: symbol [file]	
load	dynamically link file and add its symbols.	
dir	Adds directory path to front of source path or clear source path. Options: dir [path]	dir c:/slos dir
list	Shows source lines. Options: list               ; list next 10 lines of source list -           ; list previous 10 lines list s,e       ; list lines from s to e list *addr   ; line line containing addr	list list – list 10,50 list main.c:main

	list [file:]{num func} ; list source line num or func	
--	--	--

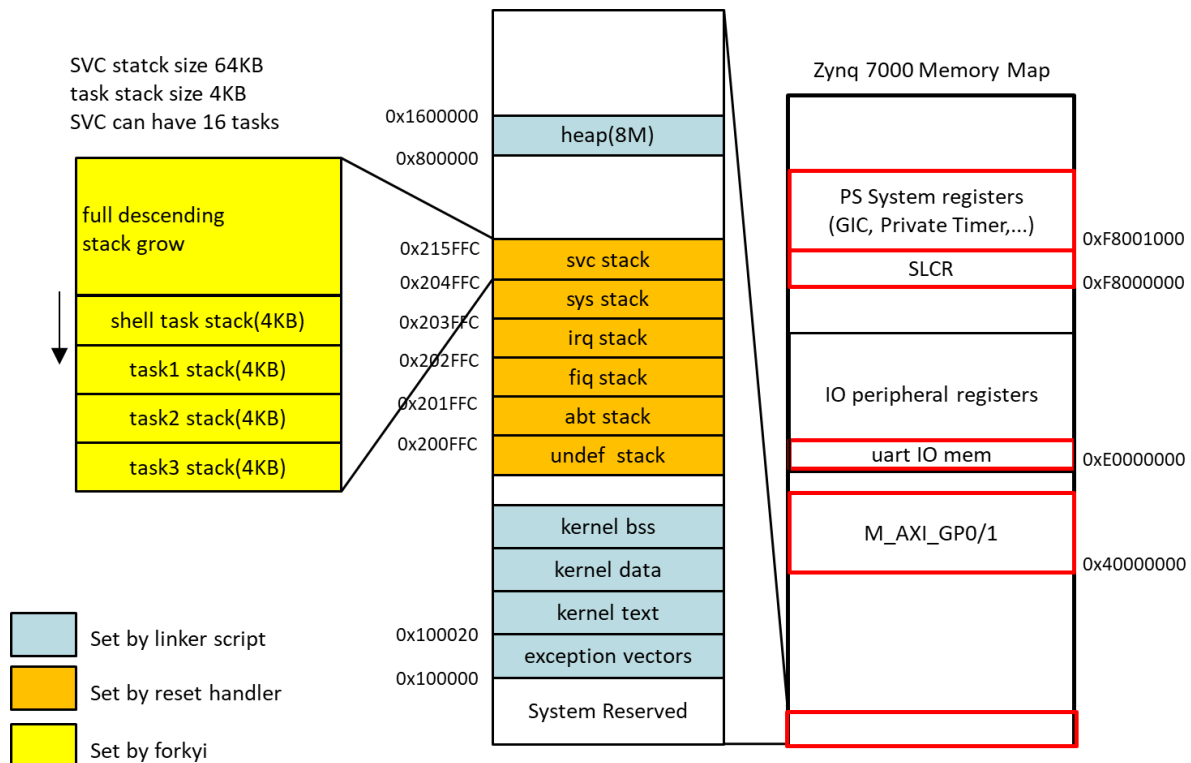
In the XMD command prompt, we can run basic command including memory read/write, register read/write, special register read etc.

## References

- [1] Linker Script : <https://sourceware.org/binutils/docs/ld/Scripts.html#Scripts>
- [2] Linker Script : [https://ftp.gnu.org/pub/old-gnu/Manuals/ld-2.9.1/html\\_node/ld\\_toc.html](https://ftp.gnu.org/pub/old-gnu/Manuals/ld-2.9.1/html_node/ld_toc.html)
- [3] BFD Library : [https://en.wikipedia.org/wiki/Binary\\_File\\_Descriptor\\_library](https://en.wikipedia.org/wiki/Binary_File_Descriptor_library)

## 4 Process Management

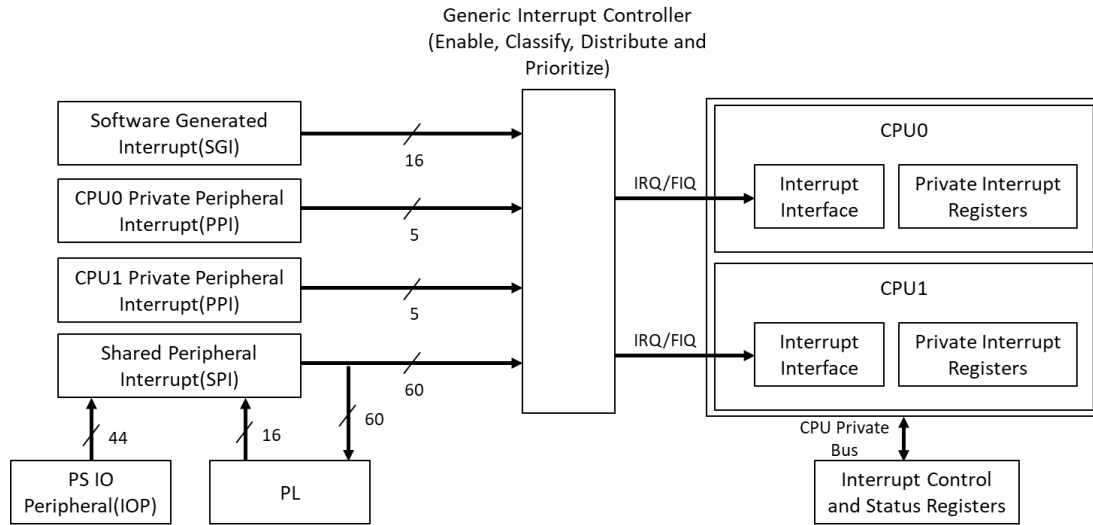
“Hello World” operating system in chapter 3 doesn’t do anything but print “hello world” message. We are going to refine this simple operating system with more advanced features. First thing is adding a scheduler for process management.



### 4.1 Exception Handler

#### 4.1.1 System Level Interrupt

Below is the system level block diagram of zynq-7000. The GIC is the interface between interrupt source hardware and the CPUs.



ARM GIC defines 3 types of interrupts; Software Generated Interrupt(SGI), Private Peripheral Interrupt(PPI) and Shared Peripheral Interrupt(SPI).

- SGI: is an interrupt generated by software writing SGI interrupt number to a ICDSGIR register and specify the target CPU(s). Zynq-7000 has 16 SGI interrupts from interrupt 0 to 15. The system uses this interrupt for interprocessor communication. In SLOS implementation, we don't use this interrupt.
- PPI: is a peripheral interrupt that is specific to a single processor. Zynq-7000 has 5 CPU private interrupts. This PPI interrupt registers are banked to each CPU.

IRQ ID #	Name	PPI #	Type	Description
26:16	Reserved	~	~	Reserved
27	Global Timer	0	Rising edge	Global Timer
28	nFIQ	1	Active low level	Fast interrupt signal from PL
29	CPU private timer	2	Rising edge	Interrupt from private CPU timer
30	AWD{0,1}	3	Rising edge	Watchdog timer for each CPU
31	nIRQ	4	Active low level	Interrupt signal from PL

Table 4.1.1: Private Peripheral Interrupts (PPI)

- We are going to use CPU private timer for timer framework. Timer framework is the base ground for process management.
- SPI: is a peripheral interrupt that the Distributor can route to any of specified combination of processor. SLOS doesn't use this interrupt.

The unique, single interrupt that SLOS need is a private timer interrupt. SLOS uses this interrupt to implement its scheduler for process management.

### 4.1.2 Reset Handler and IRQ Handler

A new kernel.S file looks like below. It was already tagged as “chapter4.1” and you can check it out by “git checkout -b your\_local\_branch\_name chapter4.1”. This kernel.S file is the entry of SLOS and let’s look into it in more detail. Current kernel.S(tagged as chapter 4.1) file has two exception vector implementations; reset handler and irq handler. We already described the ARM exception vectors in chapter 3.4. The exception vectors which is located at address 0x100000 have branch commands to proper handler routines. These exception vectors labeled “exceptions” are from line 28 to 36. Line 28 is used in linker script for designating the entry of the program. Line 29 to line 36 are composed of single ARM ISA instruction to branch to its handler routine. The syscall handler, prefetch abort handler, data abort handler, and fiq handler don’t do anything for now. You can see those implementations are just blank if you open the “kernel/core/gic.c” file. We will implement syscall handler for system call implementation and data abort handler for page translation fault in the future. For process management, reset handler and irq handler is enough.

```
1      #include "mem_layout.h"
2
3      /* arm exception code */
4      .set MODE_SVC, 0x13
5      .set MODE_ABT, 0x17
6      .set MODE_UND, 0x1b
7      .set MODE_SYS, 0x1f
8      .set MODE_FIQ, 0x11
9      .set MODE_IRQ, 0x12
10     .set I_BIT, 0x80
11     .set F_BIT, 0x40
12     .set IF_BIT, 0xC0
13     .set CONTEXT_MEM, 0x4000
14     .set SP_MEM, 0x4100
15
16     .extern platform_undefined_handler
17     .extern platform_syscall_handler
18     .extern platform_prefetch_abort_handler
19     .extern platform_data_abort_handler
20     .extern gic_irq_handler
21     .extern platform_fiq_handler
22
23     .extern main
24     .section EXCEPTIONS, "ax"
25     .arm
26     .global exceptions
27
```

```

28     exceptions :
29         b reset_handler
30         b undefined_instruction_handler
31         b syscall_handler
32         b prefetch_abort_handler
33         b data_abort_handler
34         b reserved
35         b irq_handler
36         b fiq_handler
37
38     reset_handler:
39         /*set VBAR with 0x100000 */
40         ldr    r0, =0x100000
41         mcr    p15,0,r0,c12,c0,0
42
43         /*change to supervisor*/
44         msr    CPSR_c, #MODE_SVC | I_BIT | F_BIT
45
46         /*, setup svc stack*/
47         ldr    r0,=SVC_STACK_BASE
48         mov    r13, r0
49
50         /*, Switch to undefined mode and setup the undefined mode stack*/
51         msr    CPSR_c, #MODE_UND | I_BIT | F_BIT
52         ldr    r0,=UNDEF_STACK_BASE
53         mov    r13, r0
54
55         /*, Switch to abort mode and setup the abort mode stack*/
56         msr    CPSR_c, #MODE_ABT | I_BIT | F_BIT
57         ldr    r0,=ABT_STACK_BASE
58         mov    r13, r0
59
60         /*, Switch to SYS mode and setup the SYS mode stack*/
61         msr    CPSR_c, #MODE_SYS | I_BIT | F_BIT
62         ldr    r0,=SYS_STACK_BASE
63         mov    r13, r0
64
65         /*, Switch to IRQ mode and setup the IRQ mode stack*/
66         msr    CPSR_c, #MODE_IRQ | I_BIT | F_BIT
67         ldr    r0,=IRQ_STACK_BASE
68         mov    r13, r0
69

```

```

70      /*; Switch to FIQ mode and setup the FIQ mode stack*/
71      msr   CPSR_c, #MODE_FIQ | I_BIT | F_BIT
72      ldr   r0,=FIQ_STACK_BASE
73      mov   r13, r0
74
75      /*; Return to supervisor mode*/
76      msr   CPSR_c, #MODE_SVC
77
78      /*; jump to main */
79      b     main
80      ;
81  undefined_instruction_handler:
82      b     platform_undefined_handler
83      ;
84  syscall_handler:
85      bl    platform_syscall_handler
86      ;
87  prefetch_abort_handler:
88      b     platform_prefetch_abort_handler
89      ;
90  data_abort_handler:
91      bl    platform_data_abort_handler
92      ;
93  reserved:
94      b     .
95      ;
96  irq_handler:
97      stmia r13, {r4-r6}
98      mov   r4, r13
99      sub   r5, lr, #4
100     msr   cpsr_c, #MODE_SVC | I_BIT | F_BIT /* irq/fiq disabled, SVC mode */
101     mov   r6, #CONTEXT_MEM
102     str   r5, [r6], #4 /* save return addr */
103     str   lr, [r6], #4 /* save current task lr */
104     str   r13, [r6], #4 /* save sp */
105     stmia r6!, {r0-r3}
106     mov   r1, r6
107     ldmia r4, {r4-r6} /* restore r4-r6 */
108     stmia r1!, {r4-r12}
109     mrs   r5, spsr
110     str   r5, [r1], #4 /* save spsr */
111     bl    gic_irq_handler

```



```

112      ;
113      mov    r12, #CONTEXT_MEM
114      add    r12, r12, #4
115      ldr    r14, [r12], #4
116      ldr    r13, [r12], #4
117      ldmia  r12!, {r0-r11}
118      add    r12, r12, #4
119      ldr    r12, [r12]
120      msr    cpsr_c, #MODE_SVC
121      mov    r12, #CONTEXT_MEM
122      ldr    r12, [r12]
123      mov    pc, r12
124      ;
125
126      fiq_handler:
127          bl    platform_fiq_handler
128      ;
129      .end

```

### 4.1.3 Reset Handler

Reset handler has offset 0 from 0x100000 and is the first executed instruction. This entry is set by linker script with ENTRY command. Reset handler of SLOS does two basic things which are setting exception vector base address and stack address of each processor mode. Line 39 to line 41 is for setting the VBAR (Vector Base Address) which is 0x100000. This VBAR provides the base address for exceptions that are described in chapter 2.2.4. To access the VBAR, the following two lines are used.

```

MRC p15, 0, <Rd>, c12, c0, 0          ; Read VBAR Register
MCR p15, 0, <Rd>, c12, c0, 0          ; Write VBAR Register

```

A coprocessor is a computer processor used to supplement the functions of the primary processor (the CPU). Operations performed by the coprocessor may be floating point arithmetic, graphics, signal processing, string processing, encryption or I/O Interfacing with peripheral devices. By offloading processor-intensive tasks from the main processor, coprocessors can accelerate system performance. Coprocessors allow a line of computers to be customized, so that customers who do not need the extra performance don't need to pay for it [1]. ARM can have 16 coprocessors which are CP0~CP15. CP10, CP11, CP14 and CP15 coprocessor numbers are reserved:

```

CP10: VFP control
CP11: VFP control
CP14: Debug and ETM control
CP15: System control

```

CP15 coprocessor plays important role in virtual memory management in chapter 5. The PL subsystem in Zynq-7000 can be programmed as a custom coprocessor to perform particular tasks. There is a full list of CP15 registers in section B3.17.2 in ARM architecture reference document [2].

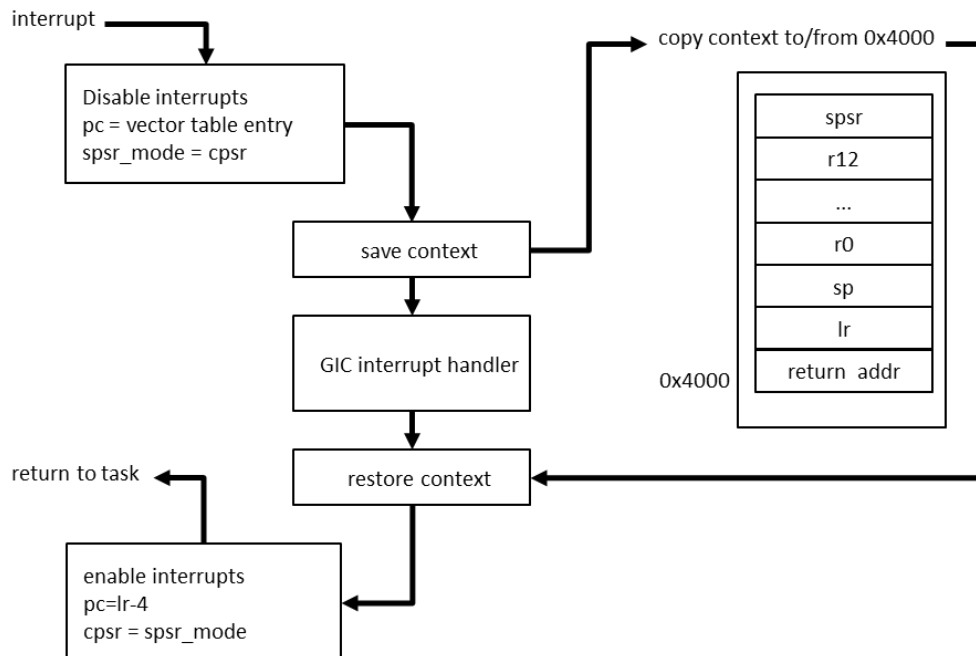
The other thing done in reset handler is setting the stack address of each processor mode. In chapter 2.2.5, the register 13(stack pointer) is a banked register of each processor mode; each mode has its own stack address. These stack addresses are set in reset handler. Line 43~73 are settings of stack address. First, it changes the processor mode properly and set the stack pointer with the value designed in figure-???.

Each mode in SLOS has 4KB stack size except supervisor mode. Supervisor mode has 64KB stack which is shared with all kernel tasks. So, supervisor mode has 16 tasks in total. Since SLOS always runs in supervisor mode, the maximum number of kernel task is 16. I think 16 max task is enough for this simple OS, but you can increase this number by redesign the memory map.

After finishing all these, the reset handler jumps to kernel main entry point.

#### 4.1.4 IRQ Handler in SLOS

SLOS irq handler has 3 parts. Register saving routine, IRQ service routine, and register restoring routine. In register saving routine, SLOS changes its mode to supervisor mode and saves all registers from address 0x4000 /\* OCM region ??? \*/. This prepares a context switch which might occur based on scheduler's decision. The second part in irq handler is jumping to IRQ service routine. IRQ service routine runs a corresponding interrupt services from pre-registered interrupt service routine vectors. Interrupt service routine is an array of function pointers of each interrupt number. The return part restores the all saved registers and goes back to the interrupted location. /\* mention that context save & restore \*/



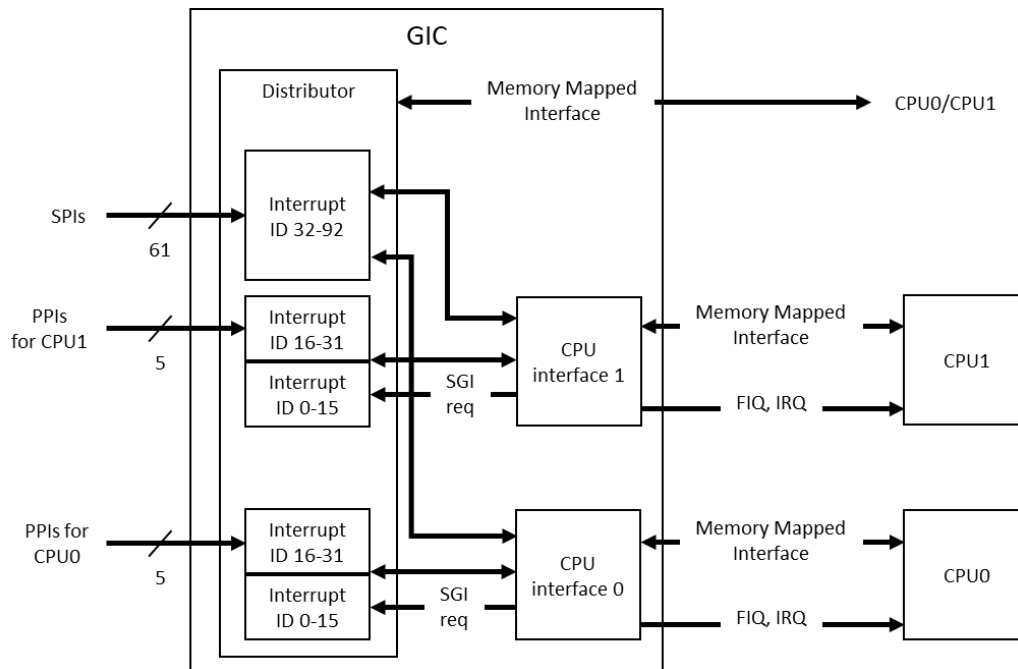
## 4.2 Generic Interrupt Controller (GIC) Implementation

GIC is the interrupt controller hardware.

### 4.2.1 GIC Partitioning

The GIC architecture splits into a Distributor block and one or more CPU interface blocks.

- Distributor: This block does interrupt prioritization and distribution to CPU interface blocks that connect to the processors.
- CPU interfaces: This block does priority masking and preemption handling of a connected processor.



### 4.2.2 Interrupt State and Handling Sequence

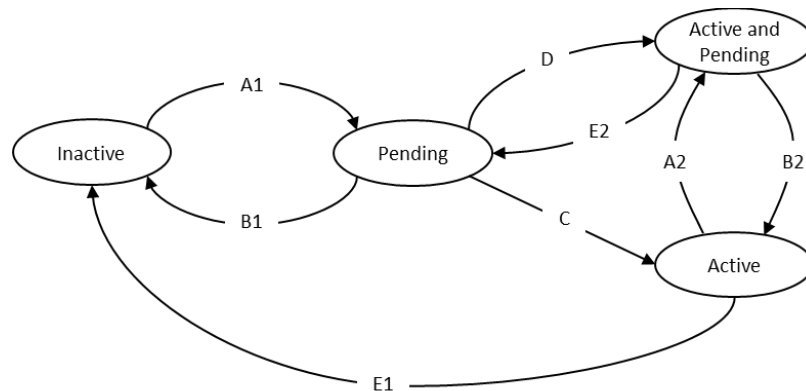
- Inactive: An interrupt that is not active or pending.
- Pending: An interrupt from a source to the GIC that is recognized as asserted in hardware, or generated by software, and is waiting to be serviced by a target processor.
- Active: An interrupt from a source to the GIC that has been acknowledged by a processor, and is being serviced but has not completed.
- Active and Pending: A processor is servicing the interrupt and the GIC has a pending interrupt from the same source.

Interrupt handling sequence is

1. The GIC determines the interrupts that are enabled. An interrupt that is not enabled has no effect on the GIC.

2. For each pending interrupt, the GIC determines target the processor or processors.
3. The Distributor forwards the highest priority pending interrupt to the targeted CPU interface.
4. Each CPU interface determines whether to signal an interrupt request to its processor, and if required, does so.
5. /\* add SLOS interrupt handling routine explanation of 4.1.4 here \*/
6. The processor acknowledges the interrupt, and the GIC returns the interrupt ID and update the interrupt state.
7. After processing the interrupt, the processor signals *End of Interrupt*(EOI) to the GIC.

The GIC maintains an interrupt state machine for each supported interrupt on each CPU interface.



When interrupt forwarded by the Distributor and target processor is signaled by CPU interface, the conditions for each state transition is as follows. These are for SPI interrupts. For SGI interrupt and for more details, refer [3].

- Transition A1 or A2, add pending state:  
A peripheral asserts an interrupt request signal or a software write to an ICD\_ISPRn register(Interrupt Set Pending register).
- Transition B1 or B2, remove pending state:  
The source signal of interrupt(level, edge) is deasserted, pending state is cleared, or a write to an ICD\_ISPRn, and a software writes to the corresponding ICD\_ICPRn(Interrupt Clear Pending register).
- Transition C, pending to active:  
If the interrupt is enabled and of sufficient priority to be signaled to processor, this transition occurs when the software reads from the ICCIAR register(Interrupt Acknowledge register).
- Transition D, pending to active and pending:  
This transition happens when all these three conditions are applied.
  - 1) The interrupt is enabled.
  - 2) Software read ICDIAR. This read adds active state to the interrupt.
  - 3) Either the interrupt signal is still asserted in level-sensitive interrupt, or a reassertion of the interrupt in the edge-sensitive interrupt.
- Transition E1 or E2, remove active state:

This occurs when the software deactivate interrupt by writing to ICCEOIR register (End of Interrupt register).

### 4.2.3 GIC Implementation

GIC initialization routines are in kernel/core/gic.c. The GIC initialization is done in gic\_init( ) function and it is called from kernel main entry function. main function in tag “chapter4.1” does two things; GIC initialization and timer framework initialization. The gic\_init( ) function initializes the Distributor and CPU interfaces.

The Distributor initialization does following steps.

- 1) Disabling GIC by writing ‘0’ to ICDDCR register (Distributor Control register).
- 2) Configure interrupt sensitivity by using ICDICFRn registers (Interrupt Configuration register).
- 3) Set interrupt target cpu with CPU0 by using ICDIPTRn registers (Interrupt Processor Target registers).
- 4) Disable all interrupts by using ICDICERn registers (Interrupt Clear Enable registers).
- 5) Enable GIC by writing ‘1’ to ICDDCR (Distributor Control register).

The CPU interface initialization is done by following steps:

- 1) Write priority masking value to ICCPMR (Interrupt Priority Mask register).
- 2) Configure CPU interface by using ICCICR (CPU Interface Control register).

GIC software has handler vector for each interrupt service routine. It is defined in kernel/inc/gic.h. There is a structure definition of interrupt service routine:

```
1  typedef int (*int_handler)(void *arg);
2  struct ihandler {
3      int_handler func;
4      void *arg;
5  };
6  struct ihandler handler[NUM_IRQS];
```

As you see, Interrupt Service Routine(ISR) is just a function pointer and its argument pointer. The client driver or software register its interrupt service routine by using gic\_register\_int\_handler( ) function with parameters of interrupt number, function pointer to its service routine, and pointer to its argument. Current SLOS needs only one interrupt – timer interrupt for its service routine, but we can add more interrupt service routines for other purpose.

GIC interrupt handler implementation is in function gic\_irq\_handler( ). This function implements the interrupt handling sequence described in chapter 4.2.2.

```
1  uint32_t gic_irq_handler()
2  {
```

```

3      uint32_t ret = 0;
4      uint32_t num, val;
5
6      /* ack the interrupt */
7      val = readl(GIC_ICCIAR);
8      /* cpuid is not used */
9      /*cpuid = val & 0x1C00;*/
10     num = val & 0x3FF;
11
12     if (num >= NUM_IRQS) {
13         return 1;
14     }
15
16     /*ret = handler[num].func(frame);*/
17     ret = handler[num].func(0);
18     /* clear int status bit */
19     writel(1, PRIV_TMR_INTSTAT);
20
21     writel(val, GIC_ICCEOIR);
22     return ret;
23 }

```

In line 7, it reads the ICCIAR (Interrupt Acknowledge register). The return value of reading this register is composed of CPU ID [12:10] and interrupt ID [9:0]. In line 10, it gets the interrupt number by masking the return value of ICCIAR register. Now since it gets the interrupt ID number, it can call the corresponding interrupt service routine from interrupt service vector. After finishing the interrupt service routine, it should write the value which is read in line 7 to ICCEOIR (End of Interrupt) register. These are steps 5 to 6 in interrupt handling sequence in chapter 4.2.2. The steps 1 to 4 are handled by the GIC hardware and we don't care about it after properly setting the GIC register in `gic_init( )`.

### 4.3 Timer Framework

Timer framework is a base ground for the process management of SLOS. Timer framework will supply 10 msec periodic timer tick to Complete Fair Scheduler or supply non-periodic, dynamic timer tick to Real Time Scheduler. SLOS has a simple timer framework but high level operating system has a well-defined timer framework. For example, following type of timer configurations are possible in Linux.

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

Linux uses its timer information for various purposes such as time keeping, time-of-day representation, a quantum for scheduling (sched tick), process profiling and in-kernel timers. Normally high-resolution dynamic timer ticks are used. This timer interrupt is also related to the CPU power consumption. If there are too frequent periodic timer interrupt, CPU periodically wakes up from its low power mode. Sometimes, even there is nothing to work with, CPU might wake up by this periodic interrupt and waste power. Power is very important in modern embedded system and this type of power loss is unacceptable. So, modern highly matured operating system uses dynamic timer interrupt and turns off periodic timer interrupt when the CPU goes to low power mode.

But first, we are going to implement 1 sec timer interrupt and the timer interrupt service routine will print “timer irq” message.

#### 4.3.1 Timer Interrupt Service Routine Implementation

There are two private timer interrupts in PPIs: Global Timer for the access of all Cortex-A9 processors and CPU private timers for each Cortex-A9 processor. To get the time information for process management, SLOS uses CPU private timer which is interrupt ID 29. Zynq-7000 private timer has following features.

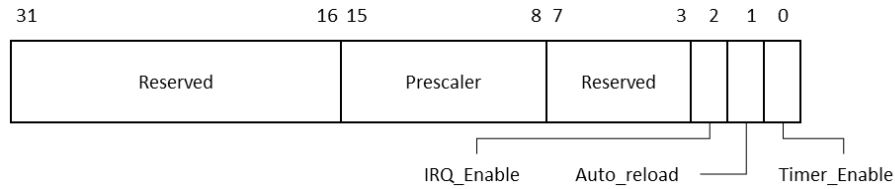
- Clock frequency is at 1/2 of the CPU frequency
- 32-bit counter that generates an interrupt when it reaches to zero
- 8-bit prescaler to enable better control of the interrupt period
- Configurable single-shot or auto-reload modes
- Configurable starting values for the counter

We can get the CPU clock frequency information from libxil/include/xparameters.h file. It is defined as `#define XPAR_CPU_CORTEXA9_0_CPU_CLK_FREQ_HZ 666666687`

xparameters.h file is auto-generated by Vivado and it describes the basic register information of the hardware designed in Vivado. Since we don’t add any custom hardware to our default design in chapter 2.6.2, the register information in xparameters.h is same with the zynq-7000 technical reference manual.

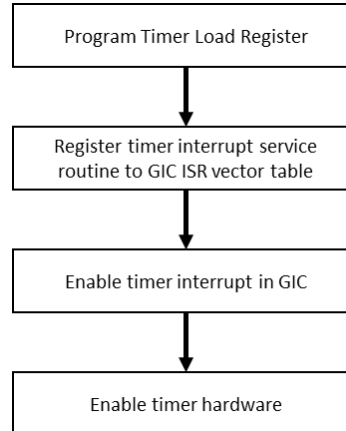
Zynq-7000 has 3 different CPU frequency in the same processor. For example, z-7020 device that is used my evaluation board has 667MHz, 766MHz and 866MHz. This CPU frequency is also related to CPU power consumption. High frequency CPU has higher power consumption than slower CPU. In our SLOS development, this CPU clock frequency is defined in xparameters.h and half of CPU clock frequency is represented as 1 second time duration.

Private timer interrupt control register is below.



- **Prescaler:** This value modifies the clock period for the decrementing event for the Counter Register.
- **IRQ\_Enable:** If set, the private timer interrupt ID 29 is set as 'pending' in the interrupt Distributor when the event flag is set in the Timer Status Register.
- **Auto\_reload:** If it is 1'b0, timer interrupt is working as single shot mode, which is if counter reaches 0, sets the event flag and stops. If it is 1'b1, timer interrupt is working as auto-reload mode, which is each time counter reaches 0, it is reloaded with the value contained in the Timer Load Register.
- **Timer\_Enable:** If it is 1'b0, timer is disabled and the counter doesn't decrement. If it is 1'b1, timer is enabled and the counter decrements.

SLOS uses auto reload mode timer. SLOS dynamically programs the Timer Load Register value based on the information of the timer framework. But in this chapter, we will program the 1 second periodic timer interrupt and implement the timer service routine. After finishing the GIC hardware initialization, the kernel main function initializes the timer hardware in `timer_init( )` function.



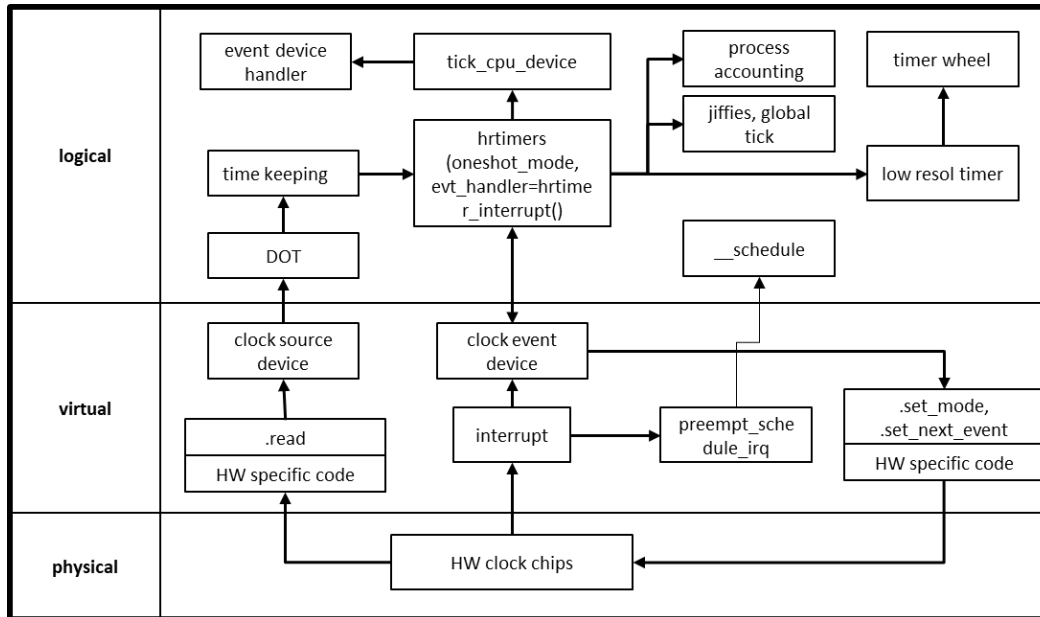
If timer interrupt in the GIC and timer hardware is properly enabled, the interrupt of private timer is periodically signaled to the CPU0. then the SLOS interrupt routine of chapter 4.1.4 is executed. The GIC interrupt handler which is described in chapter 4.2.3 figure out the type of interrupt by reading interrupt ACK register (ICCIAR Register). In this case, the timer interrupt must private timer interrupt ID #29. Then the timer interrupt service routine is called from the GIC ISR vector table. The interrupt ID #29 is used in indexing the GIC ISR table. In this chapter, the timer interrupt service routine just prints "timer\_irq" message to the console. We will fill out the timer service routine with our timer framework in next chapter. After timer interrupt service routine returns, the interrupt handler goes all the way back to the original



task. In this case the original task must be the `cpu_idle ( )`. The `cpu_idle ( )` routine is interrupted by timer interrupt every 1 second.

Now, let's follow the interrupt handling sequence with our debugger tools, GDB and XMD. Connect XMD and GDB as described in chapter 3.8.1. After the reset of processor, set the break point at 0x100018 which is the IRQ exception vector address which is the start of interrupt handling sequence. Remember that when you set the break point, you should use hardware break point, using option 'hw'. Release the processor by entering 'c' in GDB window. Then after a while the break point at IRQ vector must be hit. Then follow the sequence by using 'si' command. 'disassem' command can display the assembler code. Then run 'b gic\_irq\_handler' command that adds another break point to GIC interrupt handler start address. In the `gic_irq_handler` routine, you can check the GIC registers, CPU private timer registers by reading those register address. For this, 'p/x \*address' or 'x address' commands are used. Check the interrupt sequence implementation steps explained in chapter 4.2.2.

### 4.3.2 Quick Review on Linux Timer Framework



### 4.3.3 SLOS Timer Framework Implementation

#### 4.3.3.1 Red-Black Tree for Timer Framework

SLOS uses a red-black tree for maintaining its resources such as timer tree and run queue. We don't implement the red-black tree because it is difficult to have a stable, bugless implementation. Moreover, it is not a part of OS implementation. Thankfully, we will just borrow a good, stable and verified implementation from Linux. Red-black tree is widely used in Linux, for example Linux uses a red-black tree to manage its run queue. SLOS will use red-black tree similar as Linux does.

Even though we don't implement our red-black tree, we need to know some properties of it for better understanding of our SLOS implementation.

- Red-black tree is a one of balanced binary tree:
- Balanced tree guarantees the height of tree is  $\log n$  and searching, insert, delete operation takes  $O(\log n)$ .
- Red-black tree's insertion, removal is faster than AVL tree:
- AVL tree is another type of balanced tree. AVL tree has the difference between the height of left subtree and right subtree is 0, +1, or -1. If the height difference gets bigger than +1/-1, then AVL tree rotates the nodes for rebalancing the height. With these rotations, it maintains the balance. AVL tree has better balancing than red-black tree and this can give AVL tree better searching time. But AVL tree takes more time in rebalancing than red-black tree; red-black tree has faster insertion, removal. Insertion and removal occurs frequently in operating system. Moreover, red-black tree in Linux keeps track of the left-most child which is the searching item from scheduler. The left-most child is the next scheduled task in run queue. By maintaining this left-most child, the searching time in red-black tree takes  $O(1)$ . So red-black tree beats the AVL tree in operating system implementation.
- Red-black tree is used both in timer framework and in run queue of CFS scheduler:
- SLOS get red-black tree implementation from Linux and applies it to timer framework and run queue of Complete Fair Scheduler(CFS). SLOS also keeps track of left-most child for better performance.

#### 4.3.3.2 SLOS Timer Framework Source Tree

For the implementation of timer framework, SLOS adds following files under kernel/core.

- timer.c: This file has timer hardware initialization, timer hardware enable/disable routines, timer interrupt service routine and delay function. This file mostly involves in timer hardware.
- ktimer.c: This file has routines for managing timer red-black tree (insertion, deletion, update), sched timer creation routine, real time timer creation routine, sched timer handler, realtime timer handlers, clock source device for wall time (elapsed time).
- slmm.c: This file is for Simple & Light Memory Management. Currently, there is only one function of kcalloc (). Even it doesn't have memory free function. We will fill out this file in chapter 5 memory management. Now, this simple memory allocation is enough. This is used for timer instance allocation.
- rbtree.c: This file has implementation of red-black tree. This file comes from Linux source tree without any changes. Thanks a lot!!

#### 4.3.3.3 Timer Framework

Timer framework is running on top of red-black tree. Following is the structure of timer.

```
1 struct timer_struct {
2     uint32_t tc;
```

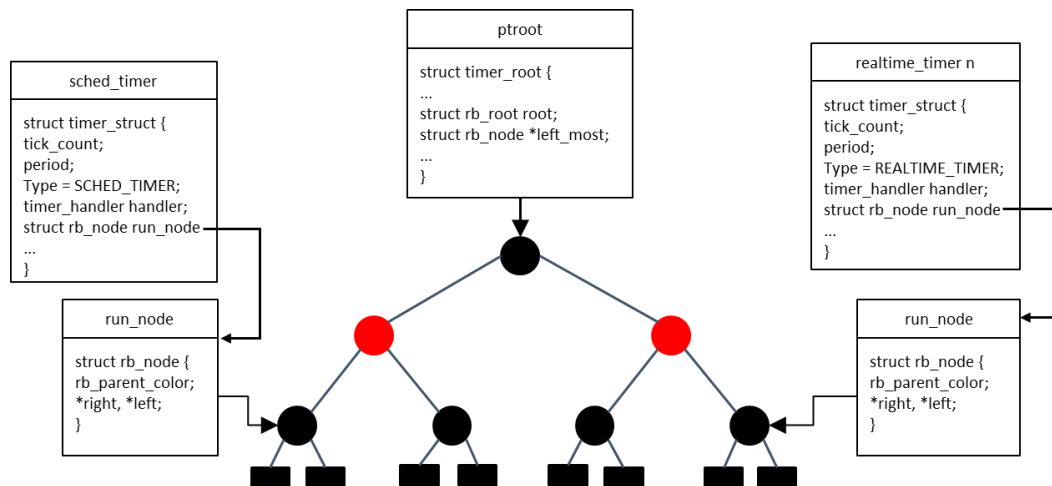
```

3      uint32_t intvl;
4      timer_handler handler;
5      uint32_t type;
6      struct rb_node run_node;
7      uint32_t idx;
8      void *arg;
9  };

```

Each timer instance has a reference (struct rb\_node run\_node) to the rb\_node in the red-black tree. Line 6 is the timer's reference element to red-black tree. 'tc' element is the tick count of timer which is the task's deadline of the scheduling. The tc value is decreased by the amount of elapsed time in timer ISR. In other words, the scheduler should run the task before the tc goes to zero. SLOS scheduler does its best effort to meet these deadlines of every timer. 'intvl' is the periodic value of the timer. The completion time of each task must be shorter than this period. The timer\_handler is a pointer to a handler function of each timer. 'type' is timer type which is one of ONESHOT\_TIMER, SCHED\_TIMER or REALTIME\_TIMER. Actually, each timer handler is called based on "Earliest Deadline First(EDF)" scheme; the left-most node in red-black timer tree is the timer which has the least remaining time to deadline. Sched timer is just another special form of realtime timer. The time span (i.e. tc value) of sched timer is shared through all the CFS tasks. This will be described later along with task scheduling. In this chapter, we should focus on the timer framework implementation; how to define the timer structure, timer tree, how to create timer and add, update the timer tree.

- How the timer tree looks like: Below figure is an example of timer tree. Sched timer and realtime timer are mixed in one timer tree. See the red-black tree is balanced tree which has the height of  $\log n$ .  $n$  is the number of rb\_nodes. This makes the insertion, deletion running fast. Notice that the root node, pthead, has a pointer to the left-most node. This left-most node is for the timer which has the earliest deadline. Timer ISR will reprogram next timer interrupt with the tick count of this timer and maintaining the left-most node makes searching earliest timer in red-black tree take constant time.
- For sched\_timer, there is a sched\_timer\_handler () for managing the sched tick. Each realtime timer has its own timer handler. In below example of timer red-black tree, each timer, whether it is realtime timer or sched timer, has a run\_node to refer its place in red-black timer tree.



- Initialization of timer red-black tree: The initialization is very easy. `timertree_init()` function does this and is called in `timer_init()`. `timertree_init()` doesn't do anything but `kmalloc` the red-black tree root.
- how to create timer: By initializing the members of timer structure, timer is created. There are three kinds of creation routine which are `create_rt_timer()`, `create_oneshot_timer()`, `create_sched_timer()`, but all of them do the same thing; take the timer handler, period, index and argument pointer as their parameters and initialize the timer structure and insert the timer's `run_node` into red-black tree. The only difference the timer's type setting. Timer creation is done by below routines.

```

1 struct timer_struct *rt_timer = (struct timer_struct *)kmalloc(sizeof(struct timer_struct));
2 rt_timer->handler = rt_handler;
3 rt_timer->type = REALTIME_TIMER;          // or SCHED_TIMER, ONESHOT_TIMER
4 rt_timer->tc = get_ticks_per_sec() / 1000 * msec;
5 rt_timer->intvl = rt_timer->tc;
6 rt_timer->idx = idx;
7 rt_timer->arg = arg;
8
9 insert_timer(ptroot, rt_timer);

```

- Insertion, deletion: Linux documentation [???] explains how to insert / delete an `rb_node` into/from red-black tree. As the document says, we need to implement our own insertion and deletion routines. SLOS timer insertion routine refers the examples of that document. SLOS timer insertion function looks like below.

```

1 void insert_timer(struct timer_root *ptr, struct timer_struct *pts)
2 {
3     struct rb_node **link = &ptr->root.rb_node, *parent = NULL;

```

```

4      uint64_t value = pts->tc;
5      int leftmost = 1;
6
7      /* Go to the bottom of the tree */
8      while (*link) {
9          parent = *link;
10         struct timer_struct *entry = rb_entry(parent, struct timer_struct, run_node);
11         if (entry->tc > value)
12             link = &(*link)->rb_left;
13         else /* if (entry->tc <= value) */ {
14             link = &(*link)->rb_right;
15             leftmost = 0;
16         }
17     }
18     /* Maintain a cache of leftmost tree entries */
19     if (leftmost)
20         ptr->rb_leftmost = &pts->run_node;
21     /* put the new node there */
22     rb_link_node(&pts->run_node, parent, link);
23     rb_insert_color(&pts->run_node, &ptr->root);
24 }

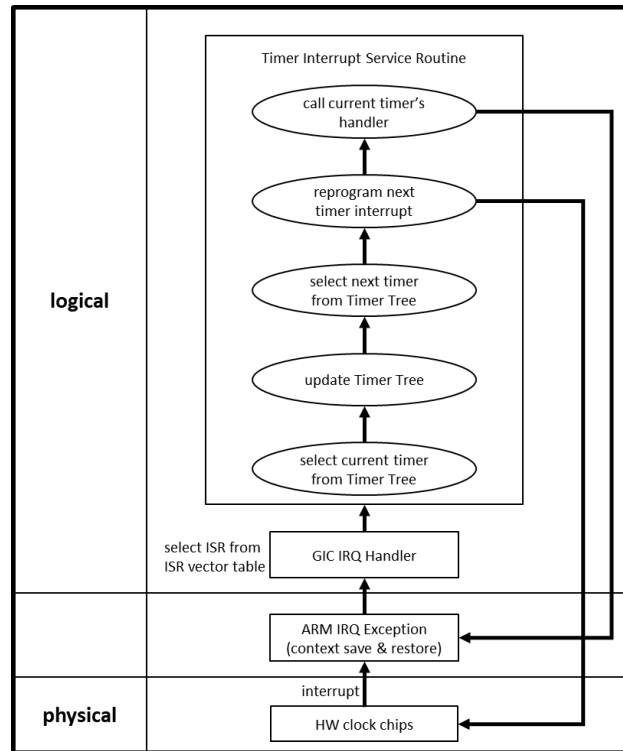
```

It gets red-black tree root and inserting timer as its arguments. It starts from root `rb_node` to the bottom on the tree by comparing the tick count of timer. Remember the tick count is the remaining deadline of the timer. If the tick count of inserting timer is lesser, then the timer has earlier deadline and it must go to the left. This is the comparison routine from line 11 to 16. Line 19~20 is for keeping the left most timer. If the inserting timer has the least value and thus it is the left most timer, update the `left_most` to point it. Line 22~23 is for rebalancing the tree.

Timer deletion is simpler than timer insertion. It calls the `rb_erase()` to remove the timer node from red-black tree. The timer deletion should keep track of left most `rb_node` and if the deletion timer is left most, then update the left most `rb_node` with the second left most timer.

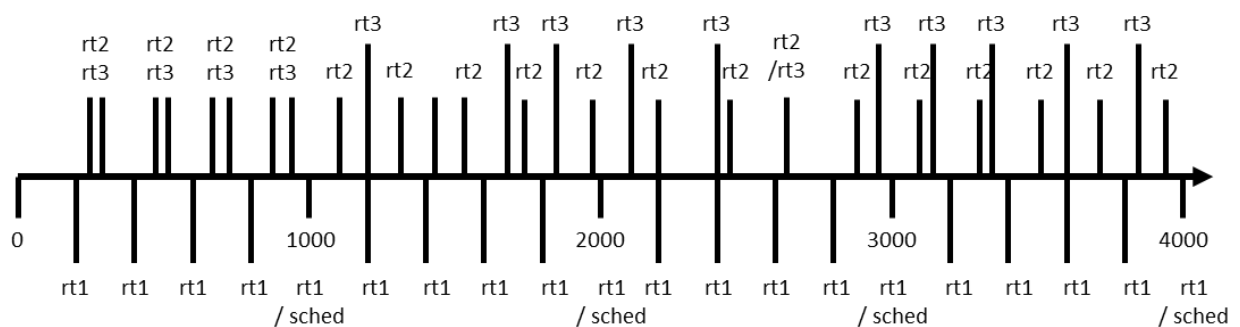
- update timer tree:

Timer ISR first gets the left-most `rb_node` in timer tree to get the timer handler of current timer interrupt. Next, timer ISR gets the elapsed time by reading the private timer load register. It contains the elapsed counter value of timer interrupt. With the elapsed time, timer ISR updates the tick count of all timers in the timer's red-black tree. Updating the tick count is done simply by subtracting the current `tc` value with the elapsed time value. After updating the timer tree, the left-most child is the next urgent timer and must be served exactly after its tick count has passed. Thus, ISR must program the timer interrupt of private interrupt hardware with the tick count of this left-most child. Finally, ISR calls the current timer's ISR handler function. This sequence is described as below figure.



#### 4.3.4 SLOS Timer Framework Test

We will test our timer framework by adding predefined timers in the timer tree. Those are sched\_timer which has 1 second timer interval, and 3 realtime timers whose timer intervals are 200msec, 220msec, and 240msec. When the timer's handler is called, it will print timer's name. We can guess the timer ISR should run below way.



In order to create test timers, following two lines are added into timer\_init().

```
create_sched_timer(sched_timer_handler, 1000, 0, NULL);
create_rt_timers();
```

create\_sched\_timer() creates sched timer which has 1000msec timer interval. 1000msec is too big for scheduler tick but it is ok for now. Currently, all the things that the sched timer handler does is printing

message. `create_rt_timers()` create test realtime timer. All these creation must be done after `timertree_init()` since adding timer means adding timer's `rb_node` into timer tree. The handler of each timer does just printing message into terminal with `xil_print()`. After creating the `BOOT.bin` image, copy it to SD card and try to boot the board. You should see the message in the described sequence.

## 4.4 SLOS Process Management

SLOS Process Management is implemented on top of timer framework.

Newly added files.

### 4.4.1 Task Control Block

Task Control Block(TCB) is a data structure containing all the information of task which is ran by a processor. TCB includes processor registers, task entry point, process id, linked list node for next/prev task, and so on. Let's look at the TCB structure in detail.

```
1 struct task_struct {
2     struct task_context_struct ct;
3     task_entry entry;
4     void *arg;
5     char name[32];
6     uint32_t pid;
7     struct sched_entity se;
8     struct list_head task;
9     struct task_struct *yield_task;
10    struct list_head waitlist;
11    TASKTYPE type;
12    uint32_t missed_cnt;
13    uint32_t state;
14 };
```

This `task_struct` is everything about the TCB. The line2 is task context. Task context is a snapshot of the processor running state. It contains the values of the ARM core registers described in chapter 2.2.5.

```
1 struct task_context_struct {
2     uint32_t pc;
3     uint32_t lr;
4     uint32_t sp;
5     uint32_t r[13];
6     uint32_t spsr;
7 };
```

These registers are the snap shot of the processor state when it runs a specific task. If a task keeps these state, it can restore the processor state whenever it come back to the processor. These registers are used in context switching and the order of members is not allowed to change for correct context switching.

task\_entry is the entry point of a specific task. It is a function pointer doing a meaningful work. SLOS's tasks are just a meaningless infinite while loop except the shell task. Shell task gets user commands from console terminal through UART and performs a corresponding work such as showing task statistics to the terminal.

struct sched\_entity is used for CFS(Complete Fair Scheduler) scheduler. This will be explained in chapter 4.5 in detail.

struct list\_head is a structure used for bidirectional linked list of tasks. There are two list\_head members in a task. One is a linked list for tasks. All tasks are connected with each other with the struct list\_head task. We can traverse all the tasks with this linked list. The other one is for the task list of in wait queue. SLOS has a wait queue to queue up tasks which is in TASK\_WAITING state. The wait queue will be explained in task state in chapter 4.4.3.

struct task\_struct \*yield\_task is a pointer to a preempted task. When a realtime task preempts a current running task, it should set this pointer with the preempted task and after finishing its job, it should yield the processor to the preempted task. This is described in chapter 4.6 in detail.

TASKTYPE type is a member for setting the task with CFS\_TASK, RT\_TASK and ONESHOT\_TASK. The timer ISR will call correct handler based on this information.

uint32\_t missed\_cnt is used for counting the number of missing the deadline of the task. When updating the timer tree, it should check if there is any task missing the deadline and put it to the left-most child of the timer tree.

uint32\_t state is representing the task state. It is one of TASK\_RUNNING, TASK\_WAITING, TASK\_STOP\_RUNNING, TASK\_STOP. This is explained in chapter 4.4.4.

#### 4.4.2 Task Creation

forkyi() function in SLOS is used for forking another task. All it does is allocate a memory for a new task and initializes the content of the task. Line 1 is for memory allocation of new task. SLOS has a 64KB stack in total for its kernel tasks. Since each task's stack is 4KB, SLOS can have 16 tasks. forkyi() function must allocate the correct stack address to forked task. Then, it initializes the members of task\_struct allocated before. Line13 sets the stack start address of the newly allocated task. Stack address is orderly assigned in the 64KB area based on the sequence of task creation. another important role of forkyi() function is setting the entry point of the task. When forkyi() is called, it gets a function pointer in its argument and sets the entry pointer with this function. All tasks in SLOS are running in Supervisor mode and line 16 is for this. Line 14 and 15 are meaningless and these are updated while task is switching context with other task. The rest part is for keeping the task linked list. The new task is added to the last of this task list.

```
1      pt = (struct task_struct *)kmalloc(sizeof(struct task_struct));
2
3      strcpy(pt->name,name);
```



```

4      pt->pid = pid++;
5      pt->entry = fn;
6      pt->type = type;
7      pt->missed_cnt = 0;
8      pt->se.ticks_vruntime = 0LL;
9      pt->se.ticks_consumed = 0LL;
10     pt->se.jiffies_vruntime = 0L;
11     pt->se.jiffies_consumed = 0L;
12     pt->yield_task = NULL;
13     pt->ct.sp = (uint32_t)(SVC_STACK_BASE - TASK_STACK_GAP * ++task_created_num);
14     pt->ct.lr = (uint32_t)pt->entry;
15     pt->ct.pc = (uint32_t)pt->entry;
16     pt->ct.spsr = SVCSPSR;
17
18     /* get the last task from task list and add this task to the end of the task list*/
19     last->task.next = &(pt->task);
20     pt->task.prev = &(last->task);
21     pt->task.next = &(first->task);
22     first->task.prev = &(pt->task);
23     last = pt;

```

If new task is CFS task, SLOS sets the task priority and add the task's sched entity to run queue. Thus, the CFS task creation routine looks like below.

```

1      temp = forkyi(name, (task_entry)cfs_task, CFS_TASK);
2      set_priority(temp, pri);
3      rb_init_node(&temp->se.run_node);
4      enqueue_se_to_runq(runq, &temp->se, true);

```

It first call `forkyi()` to allocate/initialize the new task, sets the priority of CFS task, and finally insert the sched entity to the run queue. Run queue for CFS task is another red-black tree and the `enqueue_se_to_runq()` function is similar operation with red-black tree in timer framework which is described in chapter 4.3.3.3. The only difference is that timer framework red-black tree is using the deadline of tick count as the key to compare but red-black tree in run queue of CFS task is using the virtual runtime as the key to compare. The details of virtual runtime and sched entity will be explained in chapter 4.5 CFS Scheduler.

A task creation routine for realtime task is similar but it doesn't need a run queue and priority setting. Instead, it needs to set the timeinterval of realtime task. This time interval is used for a deadline of the task. If the realtime task fails to finish its job before this deadline, it increases its missed deadline count. After setting the forked task correctly, it creates a realtime timer into the timer framework. Any realtime task has its own timer. Realtime task creation routine looks like below.

```

1      temp = forkyi(name, (task_entry)handler, RT_TASK);
2      temp->timeinterval = dur;
3      temp->state = TASK_RUNNING;
4      create_rt_timer(temp, dur, rt_timer_idx++, NULL);

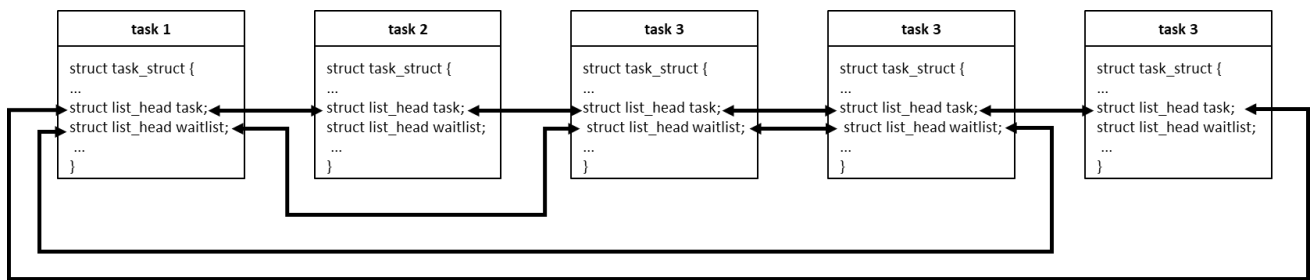
```

Another task type in SLOS is oneshot task which runs only one time. Oneshot task creation is same as realtime task except it's oneshot timer is removed in timer interrupt service routine. Removing oneshot timer is done simply by deleting its run\_node from timer red-black tree.

#### 4.4.3 Task State

Task in SLOS can have two states. Those are TASK\_RUNNING and TASK\_WAITING. CFS task can be either TASK\_RUNNING or TASK\_WAITING state. Realtime task and oneshot task have TASK\_RUNNING state only.

To change a task's state to TASK\_WAITING state, SLOS maintains a wait queue. Wait queue is a linked list of list head members of waiting tasks. task\_struct has list head members (struct list\_head wait\_list) for its linked list implementation. Below figure is an example of tasks' linked list.



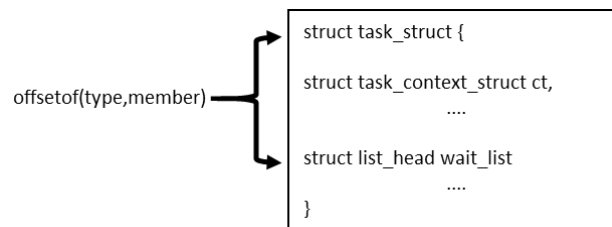
SLOS can calculate the pointer address to a task from this list\_head member by using container\_of macro. container\_of macro looks like below and is defined in defs.h.

```

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

```

container\_of macro gets the pointer to a list\_head of a task, struct type name and the member name of it. container\_of macro uses an offsetof() function C library function which calculates the offset byte of a member variable from the start of struct type.



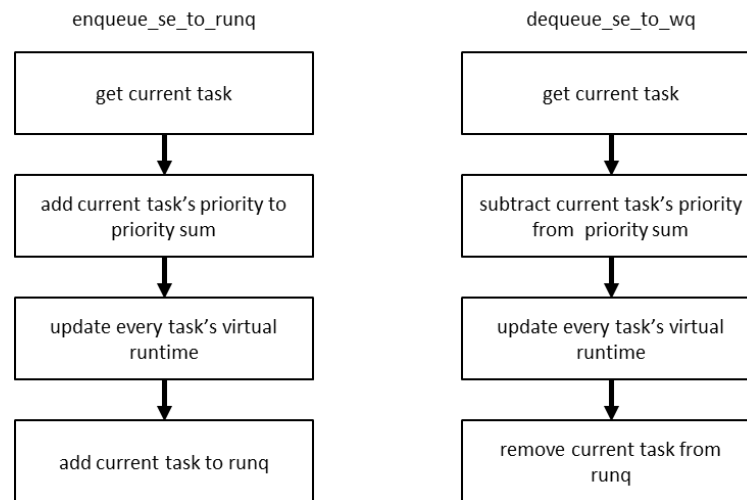
This offset is also simply calculated by below way.

```
#define offsetof(type, member) (size_t)&((type *)0)->member)
```

This `offsetof()` macro and `container_of` macro mixed with `list_head` structure is widely used in Linux. SLOS also used `container_of` and `offsetof` macros in the same way for linked list of tasks and wait queue. If we have a task's `list_head` (`task_list_head` member variable), then we can get the pointer to that specific task by

```
struct task_struct *ptask = container_of(&task_list_head, struct task_struct, task);
```

Then, wait queue is another queue for tasks whose state is `TASK_WAITING`. Changing a task's state to `TASK_WAITING` means removing that task's sched entity from run queue and sending that task to wait queue list. While changing the state of a task, SLOS needs to update each task's virtual runtime for re-evaluate the fairness of CFS tasks. `dequeue_se_to_wq()` changes a task's state to `TASK_WAITING`, and `enqueue_se_to_runq()` changes a task's state to `TASK_RUNNING`. Those two functions follow below steps.



#### 4.4.4 Context Switching

To share a CPU among multiple tasks, context switching of tasks is necessary. Each task stores its state of running into context variable. Task's running state is same as the CPU's general purpose register values. SLOS's context storage variable is `task_context_struct` structure and looks like below.

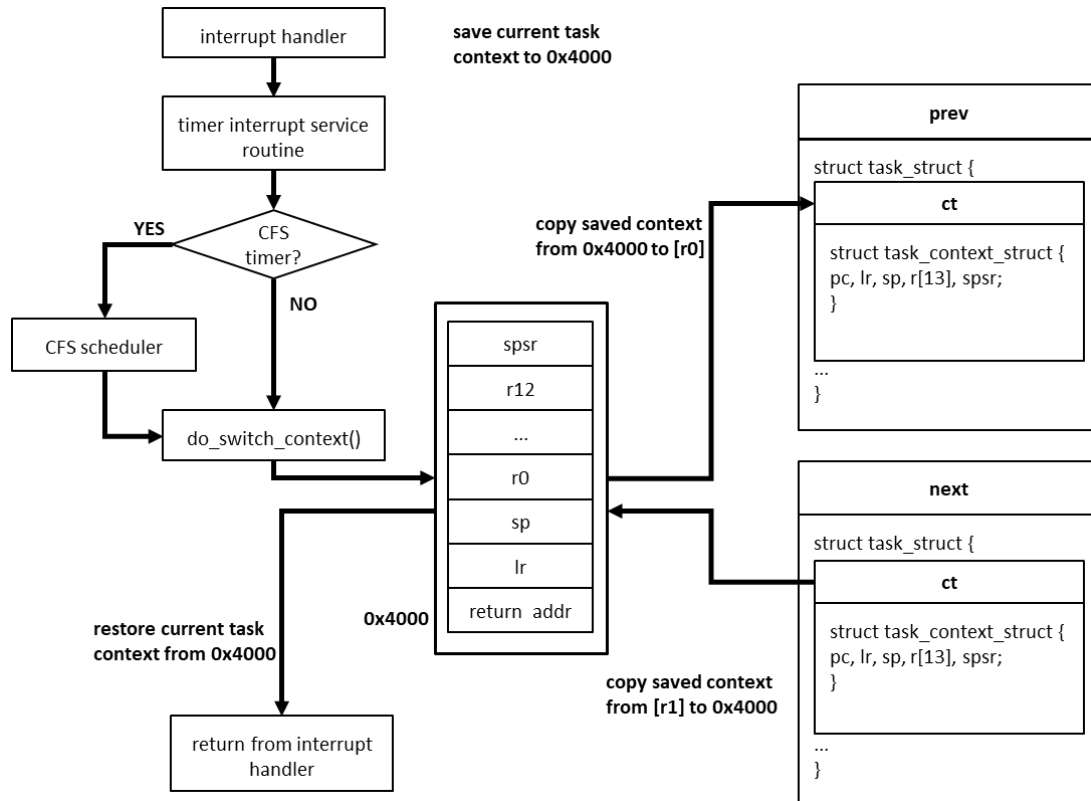
```
struct task_context_struct {
    uint32_t pc;
    uint32_t lr;
    uint32_t sp;
    uint32_t r[13];
};
```

```

uint32_t spsr;
};

```

As you can see, they are all about the cpu's general purpose registers. Thus, switching context means saving cpu registers to current task's context and restore next task's context to cpu registers. SLOS context switching happens preemptively. While current task is running, if timer framework choose another task's deadline is more urgent, then current task is preemptively switched to next task when returning from timer interrupt. The `do_context_switch(prev, next)` does switching context and is following below steps.



- 1) Timer interrupt fires and the interrupt handler saves current task's context into a specific address which is 0x4000.
- 2) After that, the interrupt handler routine calls timer interrupt handler. Timer interrupt handler will check the current timer interrupt type.
- 3) If current timer interrupt type is CFS timer, then it calls the CFS scheduler handler. If it is RT timer, then it calls `do_switch_context()`.
- 4) In CFS scheduler, it finds the worst unfair task and if it is different with current task, then it calls `do_switch_context()`.
- 5) `do_switch_context(prev, next)` routine copies the values at 0x4000 to current task's context storage variables and restores the next task's context values to 0x4000.
- 6) When returning from interrupt, the values saved at 0x4000 are restored to cpu general purpose registers and return to the next task's return address (lr value).

SLOS context switching routine is in ops.S. It is:

```
1  /* do_switch_context(struct task_struct *curr, struct task_struct *next) */
2  do_switch_context:
3      push    {r0-r4}
4      mov     r2,#0x11
5      mov     r4,#CONTEXT_MEM
6  savetxt:
7      ldr     r3,[r4],#4
8      str     r3,[r0],#4
9      subs    r2,r2,#1
10     bne     savetxt
11     mov     r2,#0x11
12     mov     r4,#CONTEXT_MEM
13  restrtxt:
14     ldr     r3,[r1],#4
15     str     r3,[r4],#4
16     subs    r2,r2,#1
17     bne     restrtxt
18     pop     {r0-r4}
19     mov     pc,lr
```

This assembler routine is composed of two parts: saving current task's context from address 0x4000(line 6 to 10), and restoring next task's context to address 0x4000(line 14 to 17). These routines are for loops copying 17 registers (return addr, sp, lr, r0~r12, spsr) of tasks' context. The do\_switch\_context() function gets its argument of the start address of previous task and next task. The task context storage is placed at the beginning address of each task and the order mustn't be changed. The register order at address 0x4000 is described in figure ???.

#### 4.4.5 Task Synchronization

Atomic execution can be achieved by interrupt disable/enable in a single core system. If interrupt is disabled, other process can't preempt the currently running process in a single core system. This method is the simplest way to implement task synchronization in single core system. But there is a problem that if current process A disables interrupt for an atomic access to a shared resource but that resource is owned by other process B which is kicked out by process A. This results in a deadlock because current process A should wait forever until the process B releases the ownership. Since the interrupt is disabled by process A, there is no way for process B to release its ownership.

There are a couple of methods to synchronize multiple tasks accessing a common resource. Semaphore, mutex, critical section, spinlock is used for this purpose. Let's look into a little more on these synchronization methods.

#### 4.4.5.1 Spinlock Implementation in SLOS

SLOS has an atomic access to shared resource by implementing the spinlock. It is done with the help of hardware synchronization method. ARM has an exclusive monitor hardware [5], and ARM ISA has commands to support synchronized load/store to a memory address. LDREX and STREX instructions are for exclusive load and store [6].

- Exclusive Monitor: An exclusive monitor is a simple state machine, with the possible states *open* and *exclusive*. To support synchronization between processors, a system must implement two sets of monitors, local and global. But for SLOS implementation, it is enough for us to know that primitive atomic operation needs the help of hardware and it is an exclusive monitor in ARM.
- LDREX: The LDREX instruction loads a word from memory, initializing the state of the exclusive monitor(s) to track the synchronization operation. For example, LDREX R1, [R0] performs a Load-Exclusive from the address in R0, places the value into R1 and updates the exclusive monitor(s).
- STREX: The STREX instruction performs a conditional store of a word to memory. If the exclusive monitor(s) permit the store, the operation updates the memory location and returns the value 0 in the destination register, indicating that the operation succeeded. If the exclusive monitor(s) do not permit the store, the operation does not update the memory location and returns the value 1 in the destination register. This makes it possible to implement conditional execution paths based on the success or failure of the memory operation. For example, STREX R2, R1, [R0] performs a Store-Exclusive operation to the address in R0, conditionally storing the value from R1 and indicating success or failure in R2.

With keeping this in mind, SLOS spinlock looks like below.

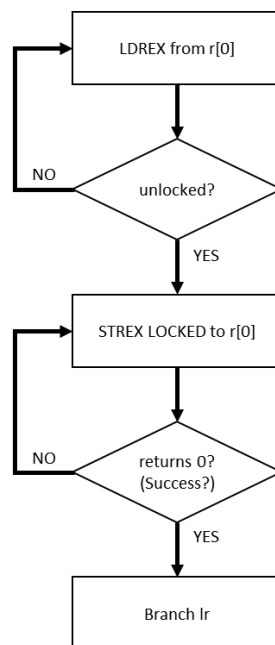
```
1  spin_lock_acquire:
2      ldr    r1,=LOCKED
3  loop1:
4      ldrexr2,[r0]
5      cmp    r2,r1
6      beq    loop1
7      /* store r1 to [r0], r2 is result */
8      strexne r2,r1,[r0]
9      cmpne  r2,#1
10     beq    loop1
11     /*lock acquired*/
12     DMB
13     bx     lr
14 .global spin_lock_release
15 spin_lock_release:
```

```

1      ldr    r1,=UNLOCKED
2      DMB
3      str    r1,[r0]
4      bx     lr

```

spin\_lock\_acquire() first loads exclusively the memory address of first argument, [r0](line 4). Then it compares the result with the value LOCKED(0x1). If they are equal, it means other process B already writes LOCKED value to this address (r[0]) before accessing common resource. Thus, current process A has to spin until the UNLOCKED is written to address r[0] by the other process B. If it is released, process A tries to store LOCKED value exclusively to address r[0] (line 8). If it fails, process B spins again until it successfully writes LOCKED value to the address r[0]. spin\_lock\_acquire() follows below steps.



spin\_lock\_release() in SLOS is simply writing UNLOCKED value to address r[0].

For synchronize multiple process, the address r[0] mustn't be in cache. So, the spin\_lock variable must be declared as a volatile variable. This is an example of how to use spin\_lock in SLOS.

```

volatile int rq_lock;
void funcA() {
    spin_lock_acquire(&rq_lock);
    /* do something useful */
    spin_lock_release(&rq_lock);
}

```

## 4.5 CFS Scheduler

SLOS has two scheduler schemes; Complete Fair Scheduler and Realtime scheduler. CFS scheduler is using a realtime timer which is periodically expired. SLOS's CFS scheduler has 10msec realtime timer. This timer quantum is shared through all CFS tasks. Fairness in CFS scheduler is according to the priority of the task. CFS scheduler measure this fairness with virtual runtime which is normalized with all CFS tasks' priority sum. The virtual runtime of a task is related to that task's priority. The virtual runtime of a high priority task is going slowly and virtual runtime of low priority task is going faster. CFS scheduler tries to balance this virtual runtime of each CFS task and picks up the most unfair task for next running task. Thus, high priority task is scheduler more often than low priority task

### 4.5.1 Run Queue, Sched Entity, vruntime, jiffies

CFS scheduler measures the time with jiffy. The `cfs_scheduler()` which is a `sched_timer`'s handler increases this value whenever `sched_timer` interrupt fires.

CFS scheduler maintains each task's fairness using `sched_entity`. `sched_entity` structure looks like below.

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

`struct sched_entity` has all quantities necessary for CFS scheduler. `jiffies_vruntime` is a virtual runtime measured with jiffy count. `ticks_consumed` is a timer tick count and `jiffies_consumed` is a measured jiffy count while the task is running. `run_node` is a node in a run queue red-black tree. `Sched_entity` holds all information needed for CFS scheduler. Each complete fair task has its own `sched_entity` as its member variable.

The measuring unit of virtual runtime is a jiffy which is increased by one every 10 msec. The virtual runtime of task A is calculated based on priority and `jiffies_consumed`.

$$virtual\_runtime_A = \frac{jiffies\_consumed_A * priority_A}{\sum priority_i}$$

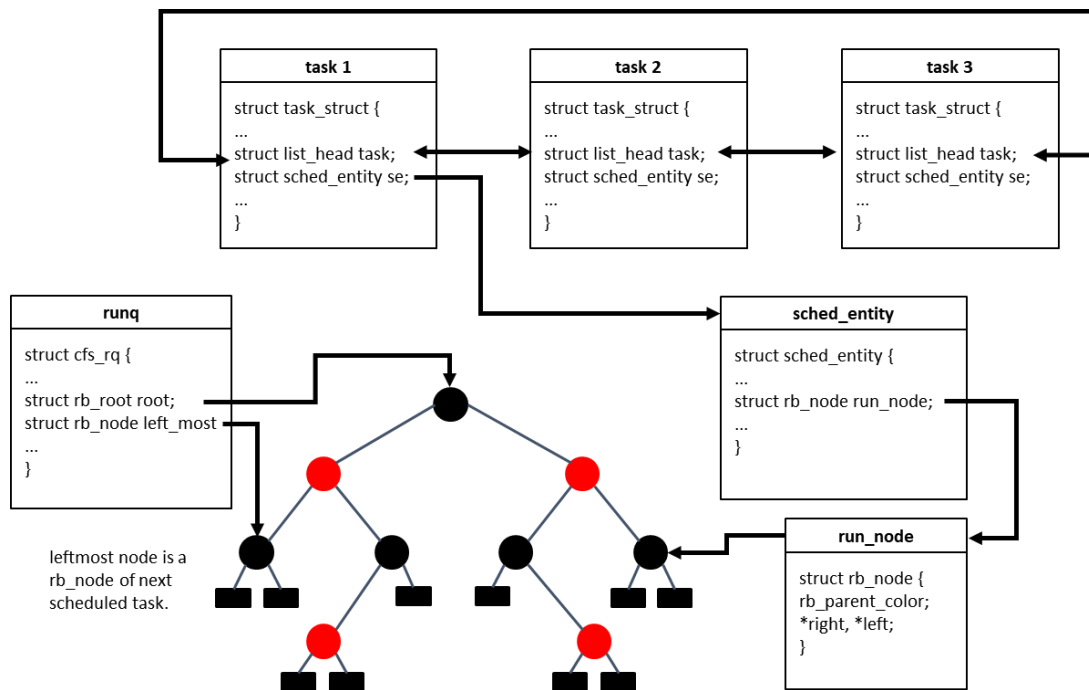
The priority of task A goes higher if the number of priority goes lower. Thus, the `virtual_runtime` of higher priority task runs slower than the `virtual_runtime` of lower priority task. For instance, if task A's priority is 2, and task B's priority is 4, then task A's virtual runtime goes 2 times slower than task B's virtual runtime does. In this case, the CFS scheduler will run task A two time more than task B to balance the fairness between them. 'Complete Fair' means fairness proportional to their priority.



All CFS tasks have its node in CFS run queue. Run queue is a red-black tree of run\_node of sched\_entity. CFS run queue has a pointer to the root rb\_node, a pointer to left most rb\_node, and a priority sum. struct cfs\_rq looks like below.

```
struct cfs_rq {
    struct sched_entity *curr, *next, *last;
    struct rb_root root;
    struct rb_node *rb_leftmost;
    uint32_t priority_sum;
    uint32_t cfs_task_num;
};
```

rb\_node of each task's sched\_entity is a node in run queue. The run queue and complete fair tasks are connected like figure ????. Notice that CFS run queue keeps the left-most rb\_node for the task waiting for next running on the cpu.



#### 4.5.2 schedule

CFS scheduler schedule () function calls switch\_context() function if current task is running too long for balancing fairness. The measurement of fairness starts from timer interrupt service routine in file timer.c. Timer interrupt service routine gets the elapsed time from timer framework. This elapsed time is measured with tick count of the timer. The current task's sched\_entity is updated with this elapsed time tick count. This is done in update\_current () in task.c.

```

1  if (current->type == CFS_TASK) {
2      jiffies++;
3      current->se.jiffies_consumed++;
4      current->se.ticks_consumed += (uint64_t) elapsed;
5      current->se.jiffies_vruntime = (current->se.jiffies_consumed) *
6          (current->se.priority) / runq->priority_sum;
7  } else if (current->type == RT_TASK) {
8      current->se.ticks_consumed += (uint64_t)elapsed;
9  }

```

This routine updates the members of sched\_entity of current task. Notice that the jiffies\_vruntime is calculated in line 5 as equation ???. If current task is realtime task, it just update the ticks\_consumed value of current task.

After updating current task's sched\_entity, the timer interrupt service routine calls CFS interrupt handler which is cfs\_scheduler() in ktimer.c. cfs\_scheduler() updates run queue with current sched\_entity. After updating the run queue, if current task's virtual runtime is not the most unfair task, then the task of left-most node in run queue must be different with current task. schedule() will check this and calls the context\_switch() which switches current task with the task of left-most node in run queue.

### 4.5.3 CPU Idle Task

The start\_kernel() function that is kernel's main entry function falls into the idle task after all initializations are finished. Normally, the idle task is not idle, it does an important job regarding power management of the processor. Since the idle task is scheduled whenever there is no job to run, it is a good time for making the processor go into its low power mode. Recently, the processor power management is very important especially in mobile devices. There are sophisticated ways to control the processor's low power modes, but we use only a ARM 'WFI'(Wait For Interrupt) command for this. So, the cpu idle is :

```

1  void cpuidle(void)
2  {
3      uint32_t i = 0;
4      xil_printf("I am cpuidle.....\n");
5
6      while (1) {
7          if (show_stat) {
8              xil_printf("cpuidle is running....\n");
9          }
10         if (i == 0xFFFFFFFF) i = 0;
11         else i++;
12         asm ("DSB" :::);

```

```

13         asm ("WFI" :::);
14     }
15 }

```

Line 12 ~ 13 is running a 'WFI' instruction. WFI instruction is used for :

- Forces the suspension of execution, and of all associated bus activity
- Suspends the execution of of instructions by the processor

The idle task tracks the activity of the bus interfaces of the processor and can signal to an external power controller that there is no ongoing bus activity. Then the power controller module regulates a proper power supply to the processor based on this information. A more detailed power management scheme is covered in chpater 4.8.

#### 4.5.4 Test of CFS Scheduler

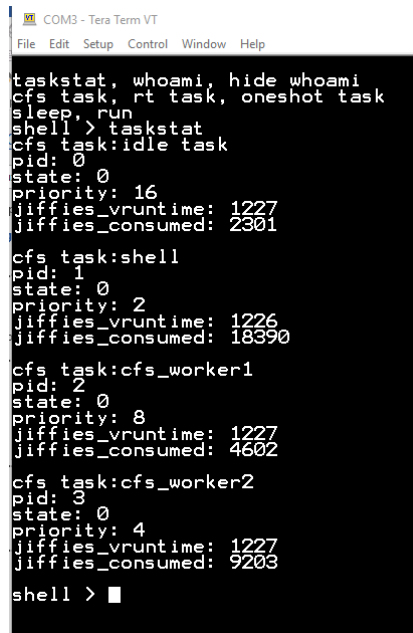
CFS scheduler assigns cpu occupation to each task based on its priority. Fairness is measured by virtual runtime determined by task's priority. SLOS tagged 4.5.3 for this chapter. Run "git checkout -b your\_branch\_name 4.5.3" from SLOS. Build the sources and package with "petalinux-package" command described in chapter ????. Basic SLOS creates two complete fair tasks in bootup sequence; cpuidle task and shell task. After finishing bootup, press enter key in serial terminal shows the shell command prompt. The shell task gets user command through this command prompt. We can see the available shell command list by entering 'help' or just hitting enter key. We can add two more complete fair tasks through this command. In the shell command prompt, type "cfs task" then shell will create two more cfs worker tasks. Let's measure the fairness of SLOS's CFS scheduler. After adding two more CF tasks, run "taskstat" command in the shell prompt. It should display all tasks' runtime information like a screen shot below. The cpu is running for 344,960msec ((2301 + 18390 + 4602 + 9203) \* 10msec) which is equal to 34,496 sched ticks. Each CF task's expected sched ticks are

- $cpuidle = 34,496 * \frac{1}{15} = 2,300$
- $shell = 34,496 * \frac{8}{15} = 18,398$
- $cfs\_worker1 = 34,496 * \frac{2}{15} = 4,599$
- $cfs\_worker2 = 34,496 * \frac{4}{15} = 9,199$

The difference with real consumed sched\_tick(jiffies\_consumed) is quite reasonable; The biggest one is under 1%. The virtual runtime is quite same through the CF tasks. This means the 'fairness' in CFS scheduler. The test result is summarized as below table.

	priority	expected cpu occupation	sched tick consumed	delta	vruntime
cpuidle	16	2,300	2,301	1	1,227
shell	2	18,398	18,390	8	1,226

cfs_worker1	8	4,599	4602	3	1,227
cfs_worker2	4	9,199	9203	4	1,227



```

taskstat, whoami, hide whoami
cfs task, rt task, oneshot task
sleep, run
shell > taskstat
cfs task:idle task
pid: 0
state: 0
priority: 16
jiffies_vruntime: 1227
jiffies_consumed: 2301

cfs task:shell
pid: 1
state: 0
priority: 2
jiffies_vruntime: 1226
jiffies_consumed: 18390

cfs task:cfs_worker1
pid: 2
state: 0
priority: 8
jiffies_vruntime: 1227
jiffies_consumed: 4602

cfs task:cfs_worker2
pid: 3
state: 0
priority: 4
jiffies_vruntime: 1227
jiffies_consumed: 9203

shell > █

```

## 4.6 Realtime Scheduler

SLOS has a soft realtime scheduling feature. For this, SLOS implements a preemptive context switching in scheduler and does its best efforts to meet the deadline. The preemptive context switching was described in chapter 4.4.4. Unlike complete fair tasks, realtime task doesn't share its time quantum with other tasks. Instead, each realtime task has its own timer tick. It must yield the cpu after completing its job to the previous task which it preempted before. The time quantum of realtime task is long enough to finish the job before the end of deadline.

### 4.6.1 yield()

Every CFS task is running through the whole time quantum. In other words, if a CFS task is scheduled to run, it occupies the cpu through the duration of sched\_tick. Since sched\_tick is 10msec in SLOS, CFS task runs at least 10msec before CFS scheduler picks another task from run queue which is suffering from unfairness.

On the contrary, the realtime task has its own timer tick in the timer framework. It can use its timer quantum for its own job only, but it should finish its job before the end of timer duration. If it fails to complete the job, then it missed the deadline and increases its missed count by one. So the timer tick duration must be set correctly before adding a new realtime task. The duration must have enough margin on the work load of the realtime task.

If the realtime task successfully finish its job before deadline, it must yield the cpu to the task which is preempted by the current realtime task. To return to the preempted task, struct task\_struct has a yield\_task pointer to a preempted task. The timer interrupt service routine updates this pointer variable

when switching context between current task and next task. If next task is realtime task, the current task is pointed by the yield\_task pointer of the realtime task. Returning to the previous preempted task is done by yield() function. yield() function does switch context between current and yield\_task. switch\_context\_yield(current, yield\_task) which is called by yield() looks like below.

```

1      switch_context_yield:
2          mov    r12, r0
3          str    lr, [r12],#4
4          str    lr, [r12],#4
5          str    r13, [r12],#4
6          stmia  r12, {r0-r11}
7          mov    r12, r1
8          add    r12, #56
9          ldmda  r12, {r0-r11}
10         sub    r12, #48
11         ldr    r13, [r12],#-4
12         ldr    r14, [r12],#-4
13         mrs    r1, CPSR
14         bic    r1, r1,#IF_BIT
15         msr    CPSR_c, r1
16         ldr    pc, [r12]
17         nop

```

This routine gets argument [r0] for the address of current task and [r1] for the address of yield\_task. The first part(line 2 ~ line6) is for saving current cpu registers to current task's context variables. These are done through store instructions. The second part(line 7 ~ line12) is restoring the cpu registers from the address [r1] which is the yield\_task address. These routines are just load instructions. The remaining part is enable the interrupt by clearing the I, F bit in CPSR register.

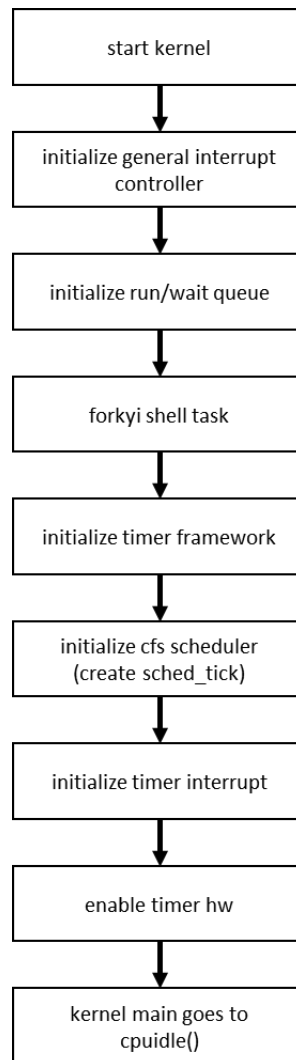
#### 4.6.2 Test Realtime Scheduler

In a shell prompt, run the "rt task" command, then shell create two additional realtime tasks. The first one has 20msec period and the second one has 25msec period. In other words, the first one should finish its job within 20msec and yield the cpu processor to the task that it preempted. The second should do the same thing within 25msec. If it missed this deadline, SLOS increase the missed count of the task. Realtime task 1 and 2 do a simple thing that increases a variable in a for loop for a moment, then yield the cpu. Check whether the task 1 and 2 miss any deadline by running "taskstat" command. Since the jobs in task 1, 2 are pretty small compared to the deadline, you shouldn't see any missed deadline.

## 4.7 Putting it altogether

### 4.7.1 Kernel Main Function Sequence

The SLOS's kernel main initializes timer framework and creates the shell task and cpuidle task. Shell task and cpuidle task are CFS task. Kernel main routine goes below sequences.



This sequence is for initialization of process management in SLOS. It will be updated in memory management later but this sequence is good enough to initialize the process management. While booting, the kernel main initialize the GIC hardware and private timer hardware. Those two subsystems are all we need to implement the process management. It also creates basic shell task by using `forkyi()`. It initializes timer framework, CFS scheduler and finally enables the timer hardware to start the process management. After completing the bootup, kernel main routine becomes cpuidle task. SLOS has only shell task and cpu idle task after bootup. We can add two more CFS tasks, two realtime task and oneshot task in a shell prompt. SLOS is a limit in stack size which is predefined in memory map describe in chapter 4 and 16 tasks

are the maximum of task number. Input “cfs task” and “rt task” after finishing boot up. SLOS process management handles the CFS tasks and realtime tasks through its timer framework. Type “taskstat” command to list up the status of current tasks. An example of the command looks like below.

```
shell> taskstat
cfs task:idle task
pid: 0
state: 0
priority: 16
jiffies_vruntime: 462
jiffies_consumed: 752

cfs task:shell
pid: 1
state: 0
priority: 2
jiffies_vruntime: 462
jiffies_consumed: 6015

cfs task:cfs_worker1
pid: 2
state: 0
priority: 8
jiffies_vruntime: 463
jiffies_consumed: 1505

cfs task:cfs_worker2
pid: 3
state: 1
priority: 4
jiffies_vruntime: 242
jiffies_consumed: 1573

rt task:rt_worker1
pid: 4
state: 0
time interval: 20 msec
deadline 0 times missed

rt task:rt_worker2
pid: 5
state: 0
time interval: 25 msec
deadline 0 times missed

rt task:rt_worker1
pid: 6
state: 0
time interval: 20 msec
deadline 0 times missed

rt task:rt_worker2
pid: 7
state: 0
time interval: 25 msec
deadline 0 times missed

shell >
```

In this example, two additional CFS tasks and four realtime tasks are added. Notice that cfs\_worker2's state is one which means it is in sleep state waiting in wait queue. Adding more CFS tasks makes the shell task respond clumsy because all CFS tasks share the sched\_tick and the more CFS tasks there are, the less sched\_tick that shell tasks can get. This results in less responsiveness of shell task.

## 4.8 Processes and Power Management

## References

- [1] <https://en.wikipedia.org/wiki/Coprocessor>
- [2] ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition
- [3] ARM Generic Interrupt Controller Architecture Specification
- [4] <https://github.com/torvalds/linux/blob/master/Documentation/rbtree.txt>
- [5] <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0008a/ch01s02s01.html>
- [6] <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0008a/CJAGCFAF.html>



## 5 Memory Management

### 5.1 Logical Address, Virtual Address and Physical Address

There are logical address, virtual address and physical address.

### 5.2 Memory Management Source Tree

SLOS has new files or has some changes in chapter 4 for memory management. The `init_mm.S`, `faults.C` are added to `kernel/exception` and the `reset_handler` and `data_abort_handler` in `kernel.S` are modified to implement the memory manager. The `init_mm.S` has the entry of secondary bootloader. The secondary bootloader initializes the MMU, translation tables, enables MMU and jump to the kernel entry. The functions in the secondary bootloader are determined by the linker script.

The `mm.c`, `vm_pool.c`, `page_table.c` and `frame_pool.c` are added to `kernel/core`. The `mm.c` has a translation table initialization function, `init_pgt()`, and heap memory allocation/free functions. The `init_pgt()` function is combined into the `.ssbl` section which is for secondary bootloader. The `mm.c` also has in implementation of `kmalloc` and `kfree` for heap memory management. The `vm_pool.c` has the implementations of virtual memory pool management. It has the virtual memory region descriptors, the implementations of allocate/release virtual memory regions. The virtual memory pool manager actually doesn't allocate the physical memory, rather it just allocates virtual memory region descriptor in its meta data page. We call it a lazy allocator. We will cover this in chapter 5.7.2. The `page_table.c` file implements the demanding pages. It initializes the kernel's page table base address and implements the address translation fault. This fault handler is used for demanding page implementation. The `frame_pool.c` file has the implementations about the physical memory management. SLOS kernel views the physical memory with 4KB frames blocks. This frame size is same as the small page size which is used by the MMU.

Another change comes from the linker script in `kernel/linker/kernel.ld`. From the memory management, SLOS uses a virtual address. SLOS is still loaded into `0x0010_0000` physically but it is loaded into `0xC010_0000` in the virtual address space. This is done in the linker script file. So, the VMA has a different address as the LMA after enabling the memory management. This is described in chapter 5.4.2.

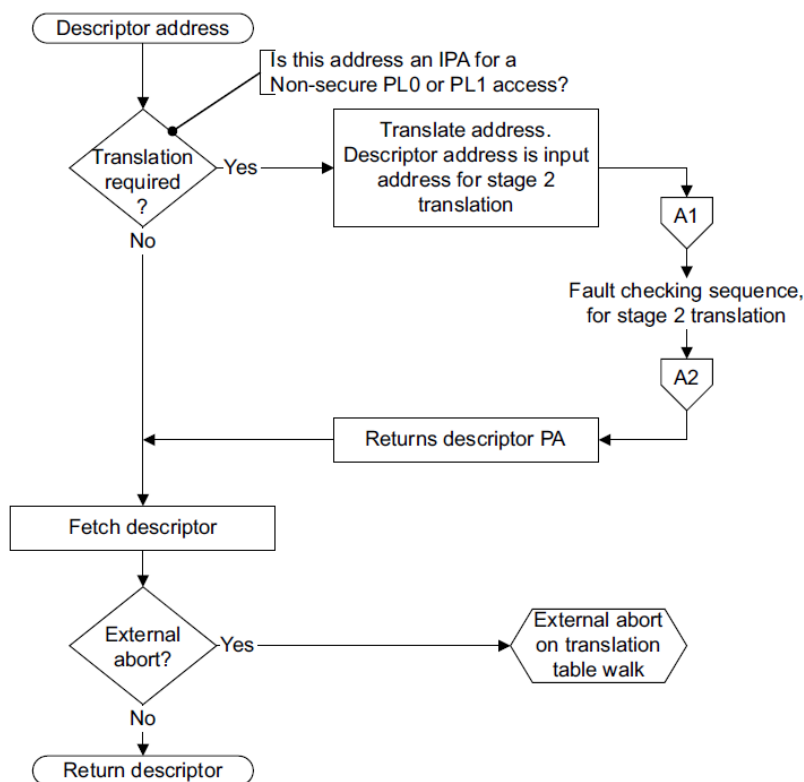
Below table shows the source changes for memory management implementations.

File Path	Description
kernel/ core/ mm.c	added virtual table initialization. <code>kmalloc/kfree</code> is changed for virtual memory allocation/free
vm_pool.c	added for region descriptor of virtual memory pool
page_table.c	added for page table management and for demanding pages
frame_pool.c	added for the management of physical memory
exception/ kernel.S	<code>data_abort_handler</code> implementation is added for demanding pages
init_mm.S	added for kernel virtual address translation tables
faults.c	added for <code>platform_data_abort_handler</code>

linker/	kernel.lids	changed for adopting the secondary bootloader and LMA/VMA separation
inc/	mem_layout.h	address definitions are changed or new definitions are added
	mm.h	header files added
	vm_pool.h	header files added
	page_table.h	header files added
	frame_pool.h	header files added

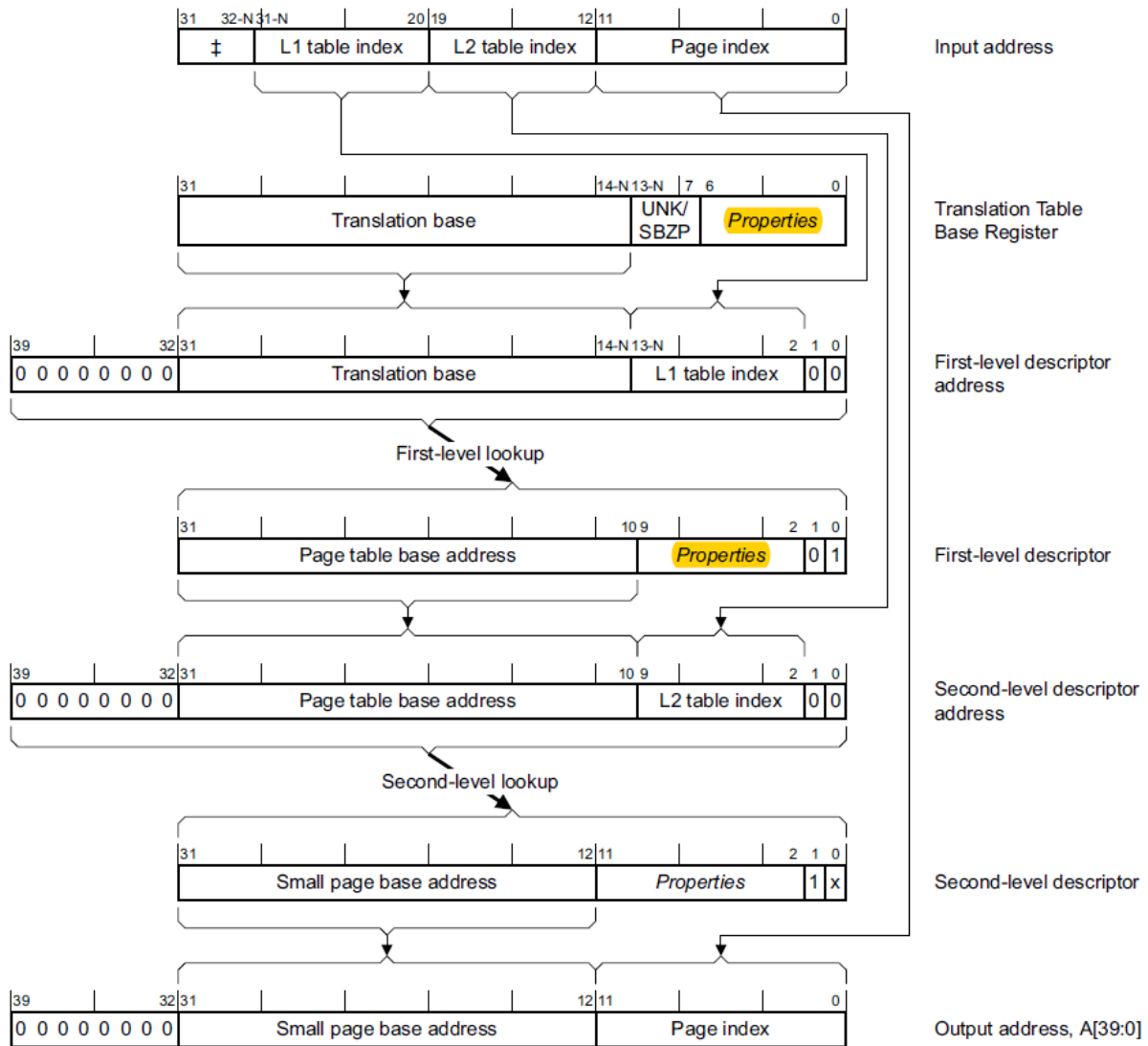
## 5.3 ARM MMU description

### 5.3.1 Address Translation in ARM MMU



**Figure B3-23 Fetching the descriptor in a translation table walk**

### 5.3.2 Small Page Table Walk



## 5.4 SLOS Virtual Memory

### 5.4.1 Linker Script Update

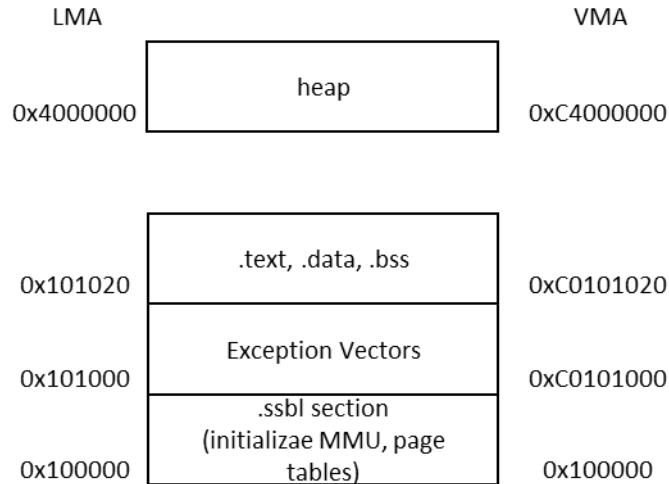
As described in chapter 3.2.1, linker script has two types of address; VMA and LMA. In chapter 4, the process management has same VMA and LMA. In chapter 5, memory management has a different VMA and LMA. The kernel text, data section location is not changed; it is located at address from 0x100000. But they will be loaded into 0xC0100000 when kernel start up. In other words, the LMA(Load Memory Address) is still 0x100000 but the VMA(Virtual Memory Address) is changed to 0xC0100000. For this, the linker script of process management in chapter 3.2.2 is changed as below.

```

1  OUTPUT_ARCH(arm)
2  ENTRY(ssbl)
3  KERNEL_HEAP_START = 0xC4000000;
4  KERNEL_HEAP_SIZE = 0x4000000;
5
6  SECTIONS
7  {
8      . = 0x100000;
9      .ssbl : {
10         *(SSBL);
11         *(PGT_INIT);
12     }
13     . = 0xC0101000;
14     .boot : AT(ADDR(.boot) - 0xC0000000) {
15         *(EXCEPTIONS);
16         *(.text);
17     }
18     .data : AT(ADDR(.data) - 0xC0000000) {
19         *(.data)
20     }
21     .bss : AT(ADDR(.bss) - 0xC0000000) {
22         *(.bss)
23     }
24     . = KERNEL_HEAP_START;
25     .kheap : AT(ADDR(.kheap) - 0xC0000000) {
26         __kernel_heap_start__ = .;
27         *(.kheap)
28         . = __kernel_heap_start__ + KERNEL_HEAP_SIZE;
29         __kernel_heap_end__ = .;
30     }
31 }

```

There are two big changes in the linker script. One is that there is another section named “.ssbl”. This section is a secondary bootloader to set up the kernel’s virtual address translation tables. The entry point of the kernel is changed to the start address of this section. The start address is still 0x100000. The second change is that the kernel start address is moved to 0xC0101000. This is a VMA. The line 13 sets the current location counter value as 0xC0101000 and kernel boot code starts from there. But since the LMA is still starts from 0x100000, the “AT” keyword is used to specify the load address of the section. The LMA can be calculated by subtracting the current LMA with offset value 0xC0000000. So, every section after .boot section sets its LMA after the section name. All sections of kernel is still from 0x100000 physically but logically it starts from 0xC0101000. Below figure describes the layers of kernel sections.

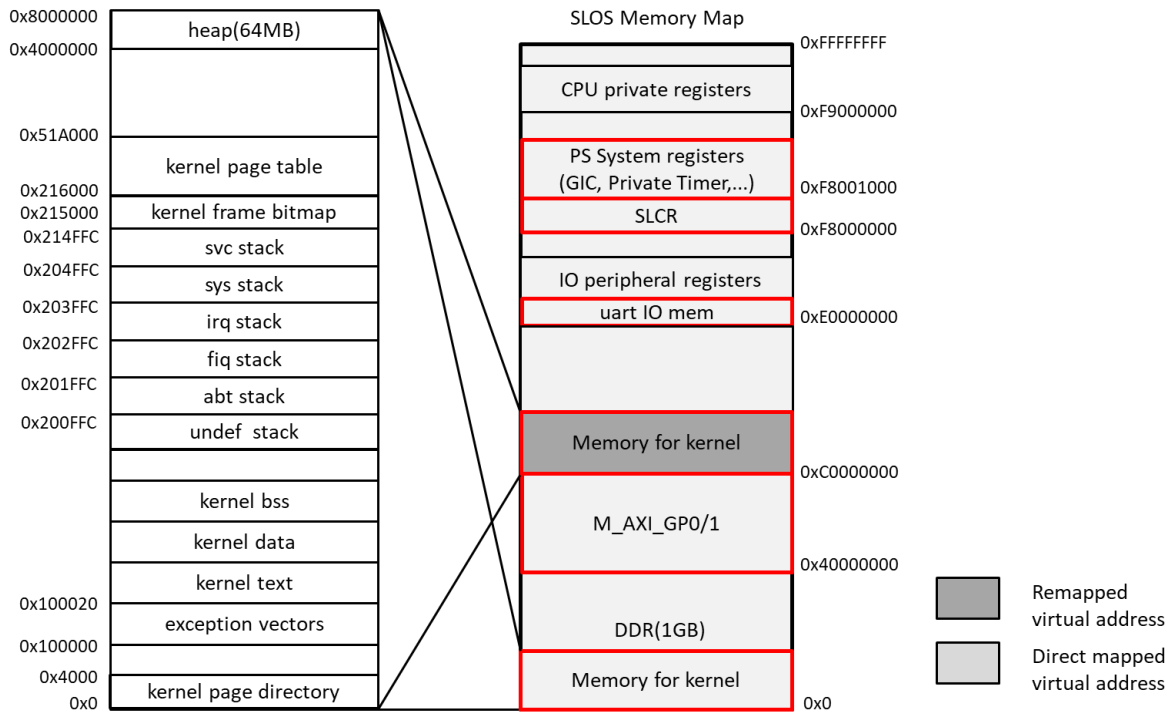


Normally, kernel is located at the high memory address and USER mode applications use a low memory address. This is because when application is built, it doesn't have an information about the memory address, it uses a default address starting from 0x0.

#### 5.4.2 SLOS Virtual Memory Map

SLOS has a virtual memory map depicted in figure ???. It has a 4GB virtual memory address range. 0x0 ~ 0xBFFFFFFF, 0xE0000000 ~ 0xFFFFFFFF regions are directly mapped address. Directly mapped address means an address whose virtual address is same as its physical address. 1GB DDR memory region, AXI interface address region, IO peripheral region, SLCR region, PS system region, CPU private registers region are all located at this address region. 0xC0000000 ~ 0xDFFFFFFF region is remapped to the physical address of 0x0 ~ 0x1FFFFFFF. This is because the kernel virtual address is started from 0xC0000000 region and this address region should be translated to the 0x0 ~ 0x20000000 physical address. So, the translation of 0x0 ~ 0x1FFFFFFF and 0xC0000000 ~ 0xDFFFFFFF are exactly same. Normally, the lower address region starting from 0x0 is assigned to the custom applications.

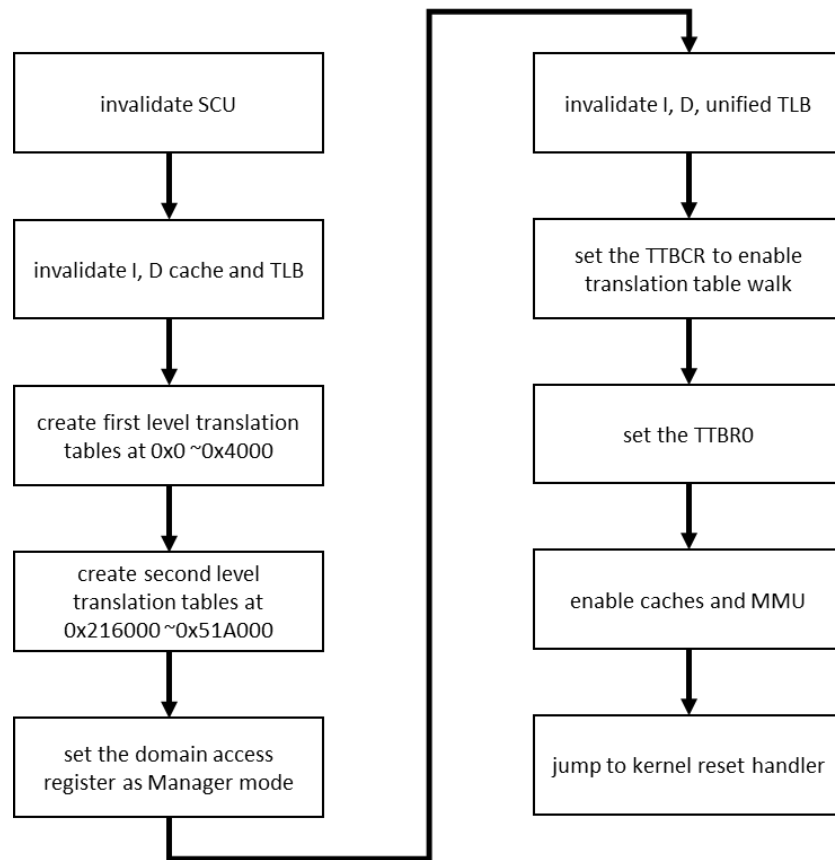
SLOS uses a small page which has 4KB page size. Since each entry in page directory table can cover 1MB physical address region, there are 4K entries in page directory table. Kernel page directories are located at 0x0 ~ 0x4000. Those are 4 pages having 4K entries for addressing the total 4GB region. Kernel start address is still 0x100000 and up to 0x215000. This region is not changed from chapter 4. Right on top of kernel stack, there is a 4KB frame bitmap for kernel frames. Kernel manages its physical memory using frames. One frame bit map which is 4KB can address 128MB physical memory and this is enough for SLOS. Page table of secondary translation is started after frame bitmap and is ranged from 0x21600 to 0x51A000. This table has 256 \* 4K entries. Kernel heap is 64MB size and starts from 0x4000000 to 0x8000000. All these regions are placed into DDR memory region.



## 5.5 Two Stage Address Translation in SLOS

### 5.5.1 Page Translation Table Initialization

SLOS uses a short descriptive format with small pages, that is, its address width is 32-bit wide and page size is 4KB. Pages need two stage translation table walk to build proper physical address. The kernel's address translation table is initialize early before starting kernel. When bootloader jumps to address 0x100000, it meets the entry of secondary bootloader which is put into .ssbl section. The secondary bootloader is composed of MMU initialization routines and page table initialization routines. Figure ??? describes the initialization flow in the secondary bootloader. For the initialization of MMU, caches, you can refer the ARM Architecture Reference Manual but it has quite too much information for our simple OS. There is a summary of CP15 register in chapter B3.7.2 in ARM reference manual. But I borrow most of these CP15 coprocessor manipulation routines from the Xilinx zync primary bootloader.



The page translation table is created in `init_pgt()` function. Since this function should be built into the `.ssbl` section, the “section” attribute keyword is used in the declaration. This function is declared as

```
void init_pgt(void) __attribute__((section("PGT_INIT")));
```

This sets the `init_pgt` function’s section name as “PGT\_INIT” and the linker script line 9 ~11 in chapter 6.3.1 will map the PGT\_INIT section to the secondary bootloader `.ssbl` section.

### 5.5.2 Initialization of First Stage Translation Table

The `init_pgt()` function first initialize the first stage translation table which has 4K entries. This table size is 16KB (4K entries \* 4B per entry), and starts from physical address 0x0 to address 0x4000. Each entry of this table covers 1MB address and 4K entry can address 4GB in total. The first entry of this table points the start address of second stage translation table. Since each entry of first translation table has 256 entries in second stage table, the address of entry in first translation table is increased by 0x400. So the lower 10bits are used for setting the property of memory. `init_pgt()` function sets this property as 0x1E1. 0x1E1 means

- Bit[1:0] = 2'b01 : 00:fault, 01:page, 10:section, 11 : reserved
- Bit[2] = 1'b0 : PXN (Privilege eXectuion Never)

- Bit[3] = 1'b0 : NS (Non-Secure)
- Bit[4] = 1'b0 : SBZ (Should Be Zero)
- Bit[8:5] = 4'b1111 : Domain 0xF
- Bit[9] = 0 : don't care

The first 2bits are for the memory block size used in the address translation.

- 2'b00: Invalid. The associated VA is unmapped. Any attempt to access that address will generate a translation fault.
- 2'b01: Page table. The descriptor gives the address of a second-level translation table, that specifies the mapping of the associated 1MByte VA range.
- 2'b10: Section or Super Section. The descriptor gives the base address of the Section or Supersection. Bit[18] determines whether the entry describes a Section or a Supersection.
- 2'b11: Section or Super Section or Reserved.

SLOS uses small page (4KB) and the Bit[1:0] is set as 2'b01. If these bits are 2'b00, the access to this address region generates a translation fault. The kernel heap region which is from 0xC4000000 to 0xC8000000 is initially not addressed, i.e. the translation table entries for kernel heap are 0x0. If there is an access to a heap address to allocate a heap memory such as kmalloc(), then it will generate a translation fault. SLOS kernel handles this by demanding pages. Anyway, SLOS's bit Bit[2:0] can be either 2'b00 for kernel heap region or 2'b01 for all other regions.

Value of Bit[4:2] is 3'b000. Bit[2] is PXN(Privilege eXecution Never) and it should be 0. Bit[3] is to set the Security Extensions and let's set it 1'b0. Bit[4] should be 1'b0.

Bit[8:5] designate the domain of memory. Domain is a collection of memory regions and Short-descriptor translation has 16 domains. The access permission to these domains are configurable through DACR(Domain Access Control Register). The DACR bit assignments are:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0																

Permission values are:

- 2'b00: No access. Any access to the domain generates a Domain Fault
- 2'b01: Client. Accesses are checked against the permission bits in the translation tables.
- 2'b10: Reserved.
- 2'b11: Manager. Accesses are not checked against the permission bits in the translation tables.

The Bit[8:5] for SLOS domain is set as 4'b1111 which is D15. Since we don't want any access permission check to the domain, all domains' access permission value is 2'b11 as manager. The CP15 can set the DACR register is set as

```
ldr    r0, =0xFFFFFFFF
mcr    p15, 0, r0, c3, c0, 0
```



Bit[9] is don't care and is set as 1'b0. Then, the property field of first translation table entry is 0x1E1. The left 32-bits of first entry is the start address of the 256 entries of second translation table. The first translation table entry is set as below pseudo code.

```

1      for i = 0 ... 4095 do
2          if i is not in kernel heap region then
3              ppage_dir[i] = (KERN_PGT_START_BASE + i * 1024) | 0x1E1
4          else
5              ppage_dir[i] = 0x0

```

KERN\_PGT\_START\_BASE in SLOS is defined as 0x21A000. Each of 4096 entries has 4 Byte size, the total size of first translation table is 16KB.

### 5.5.3 Initialization of Second Stage Translation Table

Second stage translation table points the start address of 4KB page frames in the memory. As described in memory map in chapter 6.3.2, the first 3GB(0x00000000 ~ 0xBFFFFFFF) region is directly mapped address. That virtual address of that region is same as the physical address. The 1GB DDR RAM address, master AXI interface 0/1 address fall into this region. The region property of this region is 0x472 which means

- Bit[0] = 1'b0: eXecution Never
- Bit[1] = 1'b1: 1'b0 for large page, 1'b1 for small page
- Bit[2] = 1'b0: Bufferable
- Bit[3] = 1'b0: Cacheable
- Bit[5:4] = 2'b11: AP[1:0] for R/W full access when AP[2] = 1'b0
- Bit[8:6] = 3'b001: TEX[2:0]
- Bit[9] = 1'b0: AP[2] is 0 for R/W full access
- Bit[10] = 1'b1: Shareable
- Bit[11] = 1'b0: NG(Non-Global), 0 for global

These values could be different to different usage. Now the second stage table entries for the first 3GB region can be as below pseudo code.

```

1      for i = 0 ... (3 * 1024 * 256) do
2          ppage_tbl[i] = (i * 4096) | 0x472

```

Kernel address region 0xC0000000 ~ 0xDFFFFFFF is remapped to DDR RAM address 0x00000000 ~ 0x20000000. This is because the virtual address starts from 0xC0000000 but their loaded address is in the RAM address starting from 0x00000000. This is easily done by below.

```

1      for i = 0 ... (512 * 256) do
2          ppage_tbl[3 * 1024 * 256 + i] = (i * 4096) | 0x472

```

The next 1GB address (0xE0000000 ~ 0xFFFFFFFF) is also directly mapped address except the CPU private address. The Cortex-A9 MPCore-TRM document mentions that private memory region must be marked as Device or Strongly-ordered in the translation table. For setting the property of Device or Strongly-ordered memory in the translation table, the Bit[8:6] which is for TEX[2:0] should be 3'b000. Then, the property for CPU private region (0xF8900000 ~ 0xF8F02FFF) is 0x432. The memory address for CPU private register is set by below.

```

1      j = 0
2      for i = (0xF89 * 256) ... (0xF89 * 256) + 0x602 do
3          ppage_tbl[i] = (0xF8900000 + (j * 4096)) | 0x432
4          j++

```

#### 5.5.4 MMU Programming

After creating the address translation tables, we need to set the MMU properly. The CP15 registers are the places for setup the MMU and the caches in ARM. Chapter B3.17.2 in ARM TRM document summarizes the description of CP15 registers.

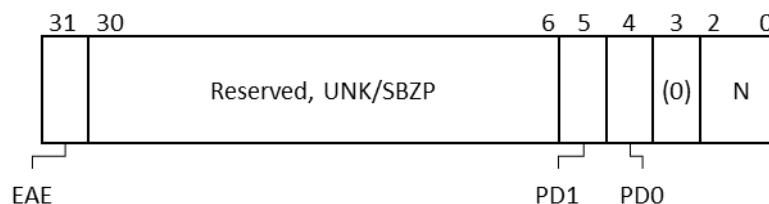
We just created the translation tables, and need to invalidate the TLBs to refresh with new table entries. Refer the ARM TRM document chapter B3.18.7 to invalidate the instruction TLB, data TLB and whole unified TLB.

```

1      /* invalidate Instruction TLB */
2      ldr    r0, =0x00000000
3      mcr    p15, 0, r0, c8, c5, 0
4      /* invalidate Data TLB */
5      ldr    r0, =0x00000000
6      mcr    p15, 0, r0, c8, c6, 0
7      /* invalidate unified TLB */
8      ldr    r0, =0x00000000
9      mcr    p15, 0, r0, c8, c7, 0

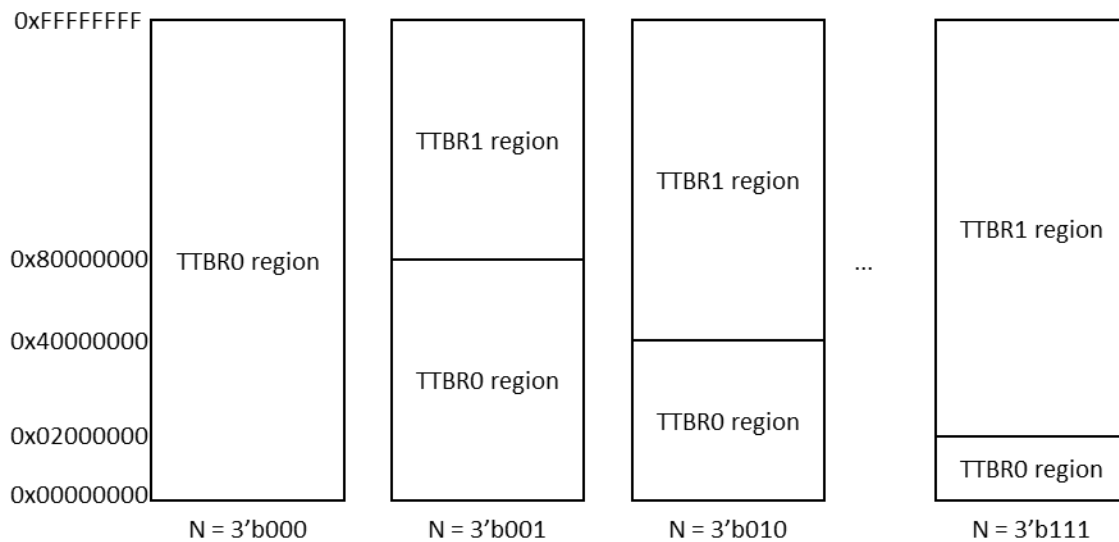
```

Next, configure the TTBCR(Translation Table Base Control Register) register. TTBCR register bit assignments in Short-descriptor translation table are:



- Bit[31]: EAE bit(Extended Address Enable). 0 for 32bit translation system with Short-descriptor translation table format. 1 for 40bit translation system with Long-descriptor translation table format.
- Bit[30:6]: Reserved, Set By Zero.
- Bit[5]: 0 for translation table walk using TTBR1. 1 for generating a Translation Fault when a TLB miss on an address translated using TTBR1.
- Bit[4]: Enable/Disable the translation table walk using TTBR0. The meaning of possible values of this bit are equivalent to those for the PD1 bit.
- Bit[2:0]: Indicate the base address held in TTBR0.
  - 1) Whether TTBR0 or TTBR1 is used as the base address for translation table walk.
  - 2) the size of translation table pointed by TTBR0.

SLOS has 32bit addressing and the EAE bit should be 0. SLOS uses only TTBR0 and doesn't use TTBR1. ARMv7 supports two different translation tables and TTBR0, TTBR1 holds the base address of translation table 0 and 1. If  $N = 0$ , then use TTBR0. If  $N > 0$ , then TTBR0 holds the lower address region whose address bits[31:32-N] are all 0 and TTBR1 holds the other upper region whose address bits[31:32-N] is not all 0. As N increases the range that TTBR1 covers is increasing like below figure.



The total 4GB address space can be split into the TTBR0 and TTBR1 and TTBR0 covers the user address space and TTBR1 covers the kernel address space. But the kernel virtual address in SLOS starts from 0xC0000000 and this scheme can't be applied to SLOS. SLOS uses only TTBR0 and value N is set as 3'b000. So, the value of TTBR0 is 0x00000000.

Next, set the TTBR0 register with the first translation table's start address which is 0x00000000. Finally enable the MMU with system control register(SCTLR) and jump to the kernel reset handler.

```

1      /* set TTBR0 as 0x0 */
2      ldr    r0, =KERN_PGD_START_BASE
3      mcr    p15, 0, r0, c2, c0, 0

```

```

4      /* read control (c1) register of cp 15 */
5      mrc    p15, 0, r0, c1, c0, 0
6      orr    r0, r0, #MASK_MMU | MASK_DCACHE
7      orr    r0, r0, #MASK_ICACHE
8      orr    r0, r0, #MASK_ACCESSPERM
9      /* enable MMU, D cache, I cache */
10     mcr    p15, 0, r0, c1, c0, 0
11     /* jump to reset handler*/
12     ldr     r0, =reset_handler
13     mov     pc, r0

```

After line 10 which is enabling the MMU, all the logical addresses are translated through the translation tables which are set in chapter 6.6.2 and 6.6.3.

## 5.6 Memory Manager in SLOS

Memory manager in SLOS composes of frame pool, page pool and virtual memory pool. Frame pool handles the physical memory and virtual memory pool handles the virtual memory pool and the page pool handles the mappings between them. Memory Manager mostly works on the heap. SLOS kernel has 64MB heap for its usage. The heap memory allocation is a lazy allocator. The allocation doesn't allocate the actual memory frames but it just updates the meta data of virtual memory pool. Then, when there is an access to the allocated memory, the memory manager will allocate the 4KB memory in the heap. This is a demanding page.

Memory manager is using frames to manage its memory. Frame is a 4KB size memory region and is a minimum size of memory allocated at a time. Frame is the minimum size of physical memory allocation. If there is a need to use 2KB heap memory, memory manager allocates one 4KB frame for that and the last 2KB memory is lost and not used. This makes an internal fragmentation.

The physical memory address is calculated as below. Parameter X is a frame index and physical address is 4KB X frame\_number.

```
#define FRAMETOPHYADDR(X) ((unsigned long)((X * 4 * (0x1 << 10))))
```

SLOS has 64MB heap from 0xC4000000 to 0xC8000000. This heap memory is split into 4KB frames and allocated by calling `kmalloc()`. If `kmalloc()` is called, memory manager adds a region descriptor to the meta data of virtual memory pool but not actually allocate the physical heap memory. The meta data of virtual memory pool is in a 4KB memory frame and is a linked list of region descriptor instances. Memory manager doesn't allocate the physical memory until there is an access to that the memory. If an access to the allocated heap occurs, then a translation fault happens because the translation table doesn't have entries for that region. The abort handler in exception vector must handle this translation fault. It will allocate a frame pool in the heap and update the translation table entry for this newly allocated region. This is a lazy allocator and is a demanding page.

### 5.6.1 Kernel Frame

Memory manager has a 4KB bitmap frame at address 0x215000. Each bit of this bitmap represents a 4KB physical frame in a memory. Then, one bitmap frame covers 128MB( $4K * 8 * 4KB$ ) memory region from address 0x0 to 0x04000000. The first half of the kernel memory is a pre-allocated region for kernel usage. This 64MB region is used for kernel text, data, stack, page tables and so on. This region should not be allocated or freed by the memory manager.

The other 64MB covers 0x04000000 to 0x08000000 is a kernel heap memory which is allocated or freed by memory manager. The virtual address for this heap region is 0xC4000000 to 0xC8000000. `kmalloc()` is used for allocating the memory in the heap and `kfree()` is for freeing that memory. This is the prototype of those functions.

```
void *kmalloc(uint32_t size);  
void kfree(uint32_t addr);
```

`kmalloc()` gets a memory size as its parameter, then returns the start address of the allocated memory. This function calls the virtual memory pool's `allocate()` function. The `allocate()` function is a lazy allocator which doesn't allocate the physical memory frame. `kfree()` gets a start address of that memory as its parameter, then frees that memory from heap.

Below figure ??? shows the kernel frames. Each square block is 4KB size frame. The layout of the 64MB frames is done from many parts such as linker script, reset handler, memory manger, forkyi, and so on. These frames are predefined and are not freed.

This kernel frames are mapped to two places in virtual memory address. First, it is direct mapped to virtual address from 0x0000\_0000 to 0xC800\_0000. Virtual address is same with its physical address in this region. Actually the first 1GB virtual address is directly mapped to its physical address. This is described in chapter 6.4.2. Second, the virtual address from 0xC000\_0000 to 0xC800\_0000 is also mapped to physical address 0x0000\_0000 to 0x0800\_0000. This is for loading kernel.elf to 0xC000\_0000. All kernel symbols are based on address 0xC000\_0000. If you run 'readelf', the symbol location is starting from 0xC010\_1000 which is set in the linker script.

0x08000000	...	...	...	...	...	kerne heap frame N
0x04000000	kernel heap frame 0	kernel heap frame 1	...	...	...	...
	unused region					
0x0021B000	kernel PGT1	kernel PGT2	...	...	...	kernel PGTN
0x00215000	kernel frame bitmap					kernel PGT0
0x00215000	UNDEF handler stack	abort handler stack	FIQ handler stack	IRQ handler stack	SYS handler stack	SVC task stacks
0x00200000	.ssbl	.boot, .data, .bss				
0x00100000						
0x00000000	PGD0	PGD1	PGD2	PGD3		

### 5.6.2 Virtual Memory Pool

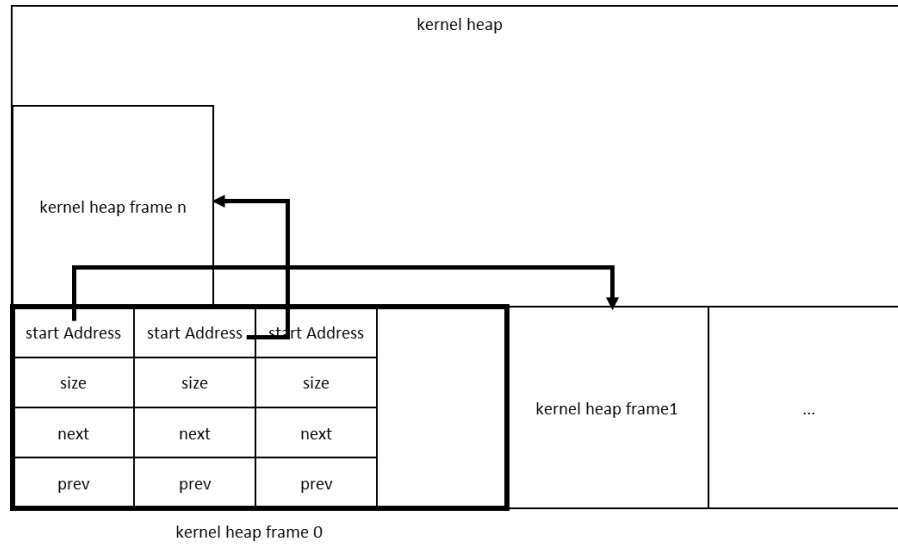
Virtual memory pool is the pool that holds the meta data of the memory requested from `kmalloc()`. `kmalloc()` calls the `allocate()` of virtual memory pool. The first kernel heap frame is assigned to this meta data. The structure of this meta data looks like below.

```

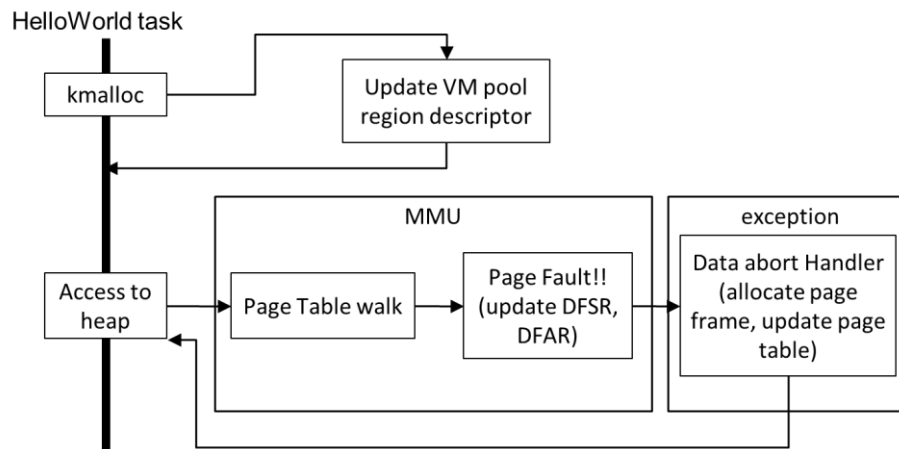
1  /* region descriptor structure */
2  struct region_desc {
3      unsigned int startAddr;
4      unsigned int size;
5      struct region_desc *next;
6      struct region_desc *prev;
7  };

```

This region descriptor has start address and its size and is doubly linked list with next region descriptor. Figure ??? is the description of region descriptor in the kernel heap frame 0.



The `allocate()` function adds another region descriptor to the end of this linked list and returns. There are no actual memory allocations at this moment. The actual memory frame allocation will be done when there is a memory access. The virtual memory pool is a lazy allocator which the real memory allocation happens when there is an access to that memory. Following figure ??? describes the concept of the lazy allocator in SLOS.



Since only kernel heap frame 0 is assigned to the container for this region descriptor, there is a limit of the number of this linked list and even there is no check routine for the limit of this region descriptor. The maximum number of region descriptor for the kernel heap is calculated as below.

$$N_{region\_descriptor} = \frac{sizeof(frame_0)}{sizeof(region\_descriptor)}$$

Nonetheless, this is still enough for SLOS's virtual memory pool.

## 5.7 Demanding Page Implementation

For demanding page implementation, we need to add translation fault handler. When the kernel tries to access an address which is not in the translation table, the data abort exception is fired. So, we need to add our demanding page routine to data abort handler.

### 5.7.1 Fault Checking Sequence

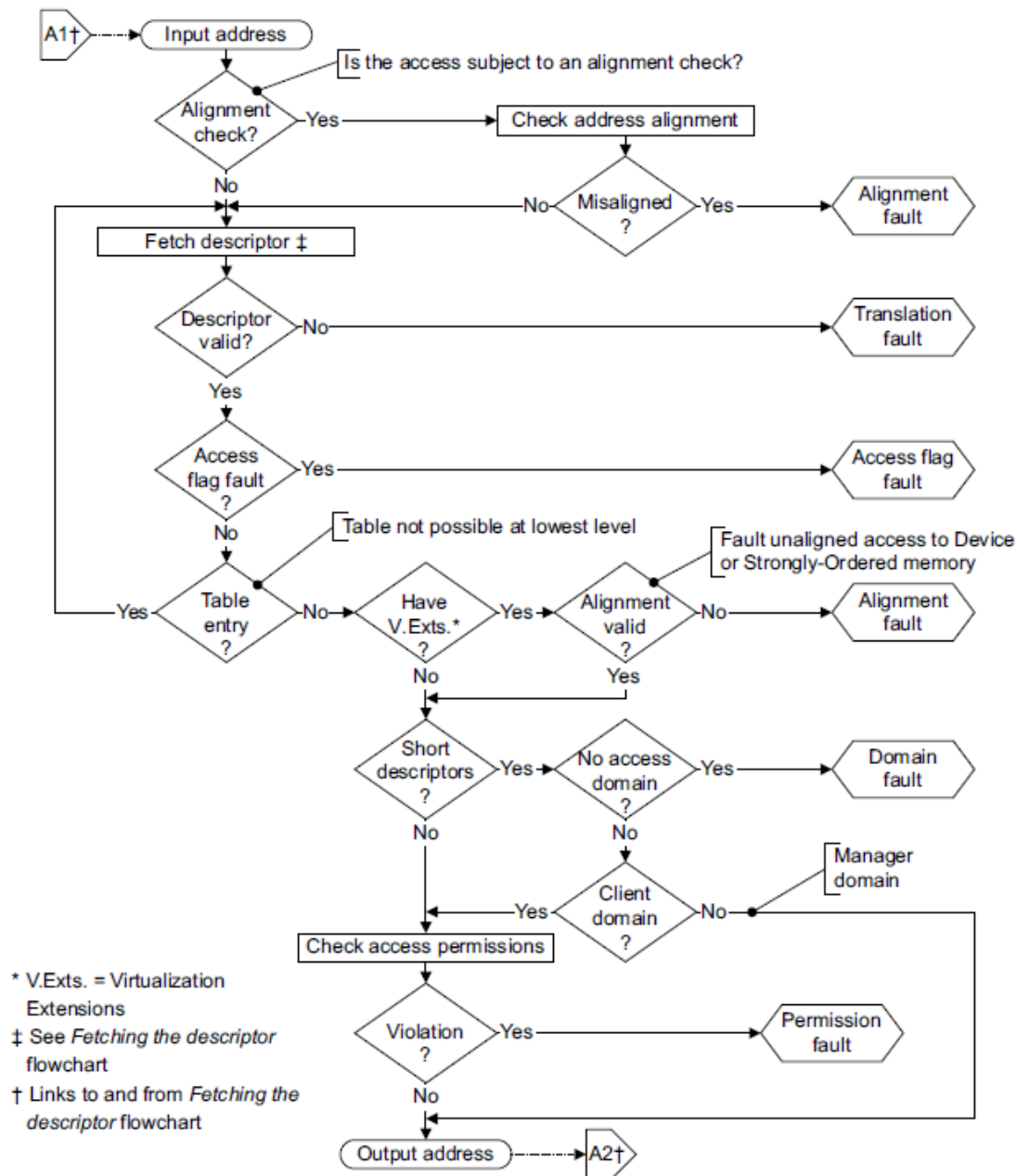


Figure B3-24 VMSA fault checking sequence



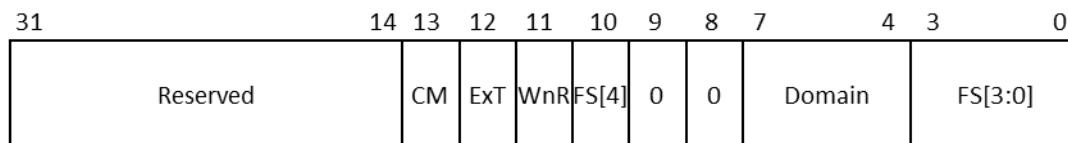
SLOS kernel supports only data translation fault.

### 5.7.2 Data Abort Handler

Now, the data abort handler in the kernel is changed as below.

```
1 data_abort_handler:
2     sub    r12, r14, #8
3     msr    cpsr_c, #MODE_SVC | I_BIT | F_BIT
4     stmfd  sp!, {r0-r11, r12}
5     push   {r14}
6     mrc    p15, 0, r0, c5, c0, 0
7     bl     platform_data_abort_handler
8     pop    {r14}
9     mrs    r0, CPSR
10    bic    r1, r0, #I_BIT|F_BIT
11    msr    cpsr_c, r1
12    ldmdf  sp!, {r0-r11, pc}
```

It subtract the return address by 8 and changes the processor mode to SVC and save the registers to the stack. The line 6 reads the DFSR (Data Fault Status Register) from CP15 and deliver this value to platform\_data\_abort\_handler. Refer the ARMv7 reference manual B3.17.1 for the assembler code of this. The DFSR holds the status information of the last data fault. Below is the DFSR register format.



- Bit[31:14]: Reserved
- Bit[13], CM: If implementation includes the LPAE, this bit means cache maintenance fault. 0 is for an abort not caused by a cache maintenance operation. 1 is for an abort caused by a cache maintenance operation. If implementation doesn't include LPAE, this bit is reserved.
- Bit[12], ExT: External abort type. In an implementation that doesn't provide any classification of external aborts, this bit is UNK/SBZP.
- Bit[11], WnR: Write not Read bit.
- Bit[10], Bit[3:0], FS: Fault Status bit. For the valid encoding of these bits, see below table ???.
- Bit[9], LPAE: Reserved UNK/SBZP.
- Bit[8]: Reserved. UNK/SBZP.
- Bit[7:4], Domain: The domain of fault address. ARM deprecates any use of this field. This field is UNKNOWN on a Data Abort exception.

We are interested in Fault Status bits because SLOS supports only translation fault. Table ??? lists the fault status bit encodings. Even in the fault status list, SLOS supports only the translation fault in MMU. So, the bold box is the only faults that SLOS implements. The first level translation fault is called section translation fault and the second level translation fault is called page translation fault.

FS	Source	Notes
00001	Alignment Fault	
00100	Fault on instruction cache maintenance	
01100	Synchronous external abort on translation table walk	First level
01110		Second level
11100	Synchronous parity error on translation table walk	First level
11110		Second level
00101	Translation fault in MMU	First level
00111		Second level
00011	Access flag fault in MMU	First level
00110		Second level
01001	Domain fault in MMU	First level
01011		Second level
01101	Permission fault in MMU	First level
01111		Second level
00010	Debug event	
01000	Synchronous external abort	
10000	TLB conflict abort	
10100	Implementation defined	
11010	Implementation defined	
11001	Synchronous parity error on memory access	
10110	Asynchronous external abort	
11000	Asynchronous parity error on memory access	

The platform\_data\_abort\_handler() in faults.c checks its first argument which has the DFSR register value and calls a proper abort handler. The platform\_data\_abort\_handler() looks like below.

```

1      if ((dfsr & TRANSLATION_FLT_PG) == TRANSLATION_FLT_PG) {
2          handle_fault();
3      } else if ((dfsr & TRANSLATION_FLT_SEC) == TRANSLATION_FLT_SEC) {
4          handle_fault();
5      } else {
6          abort();
7      }

```

This routine is checking the first argument value with a corresponding fault status masks. As you can see, there is only two checking routines for section translation fault and page translation fault. The mask values are coming from the table ???.

```
1      #define TRANSLATION_FLT_SEC          0x5
2      #define TRANSLATION_FLT_PG          0x7
```

### 5.7.3 Page Fault Handling

If platform\_data\_abort\_handler() checks the current fault type is translation fault in MMU, it will call a function handle\_fault() in page\_table. Below is the code of handle\_fault() function.

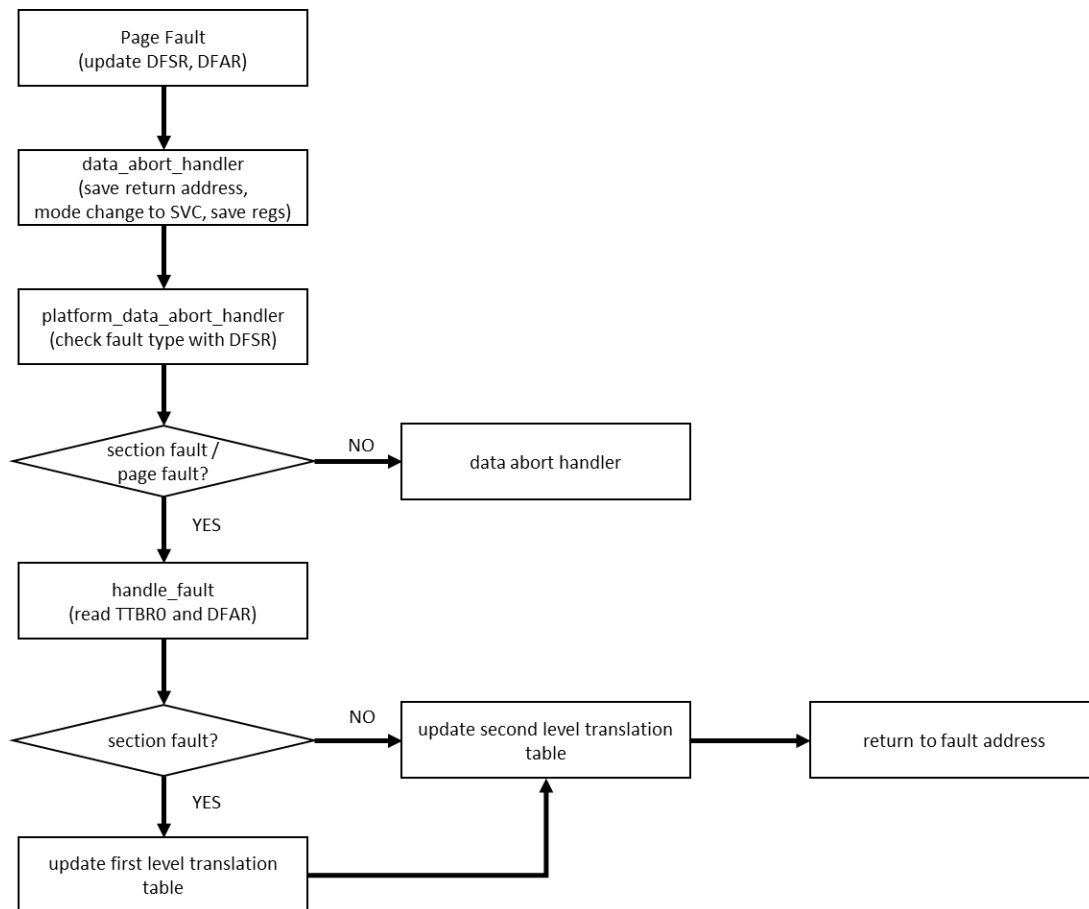
```
1      asm ("mrc p15, 0, %0, c6, c0, 0" : "=r" (pfa) ::);
2      asm ("mrc p15, 0, %0, c2, c0, 0" : "=r" (pda) ::);
3
4      pgdIdx = ((unsigned int)pfa & 0xFFF00000) >> 20;
5      pde = (unsigned int *)(((unsigned int)pda) + (pgdIdx << 2));
6
7      if ((*pde & 0x00000003) == 0x0) {
8          *pde = (KERN_PGT_START_BASE + ((pgdIdx << 8) << 2)) | 0x1E1;
9      }
10
11     frameno = get_frame(pcurrentpgt->pframepool);
12     frame_addr = (unsigned int *)FRAMETOPHYADDR(frameno);
13
14     pgtdIdx = ((unsigned int)pfa & 0x000FF000) >> 12;
15     pte = (unsigned int *) (KERN_PGT_START_BASE + ((pgdIdx << 8) << 2) + (pgtdIdx << 2));
16     *pte = ((unsigned int)frame_addr | 0x472);
```

All handle\_fault() function does is to insert a new entry for the fault address into the translation tables. For this, handle\_fault() reads the fault address in the line 1 into variable pfa. It also read a translation table base address from TTBR0 register in line 2 and save it into variable pda. Refer the ARMv7 reference manual B3.17.1 for the assembler code of this. The line 1 and 2 is using inline assembler for this. You can find inline assembler description by searching web. I used a reference [??] for this. Now we know the fault address, we can calculate the entry of it in the first translation table. This is done in line 4 and 5. Line 7 to 9 checks the last 2 bits. The description about these 2 bits are described in chapter 6.6.2. If these two bits are 2'b00, it means there is no entry in the first translation table. This is a section translation fault. Line 8 create a new section entry for the fault address. Line 11,12 gets a new memory frame from frame pool. Line 14 and 15 calculate the page entry about the fault address. Then line 16 assigns the address of allocated memory frame to this page entry.

Remember that the memory frame allocated in line 11 can be from any place in the memory. In other words, the physical memory address is independent of the virtual address. For example the virtual address

0xC400\_1000 can point physical location in 0x0400\_1000. Moreover, the other process which has its own translation tables can have the virtual address 0xC400\_1000 pointing to physical location 0x0500\_1000. By using this, each process has separate address map and has its own 4G address space. This is another advantage of using MMU and virtual address.

#### 5.7.4 Demanding Page Flow



#### 5.8 Putting It Altogether

After the memory management is implemented, the SLOS has MMU enabled and runs in the virtual address space. The heap allocation in `forkyi()` for allocating the `task_struct` of a new task is now using lazy allocation and demanding pages. In addition, the `cfs_worker1` task allocates a 1KB memory in the heap and accesses the memory for test purpose. The body of `cfs_worker1` task will be changed as below.

```

1  #define TEST_KMALLOC_SZ 4096
2  uint32_t cfs_worker1(void)
3  {

```

```

4      uint8_t *pc;
5      uint8_t t;
6      int i;
7
8      pc = (uint8_t *)kmallocc(sizeof(uint8_t) * TEST_KMALLOC_SZ);
9      if (pc != NULL) {
10         for (i = 0; i < TEST_KMALLOC_SZ; i++) {
11             t = (uint8_t)(i % 256);
12             pc[i] = t;
13         }
14     }
15
16     kfree((uint32_t)pc);
17     pc = NULL;
18
19     pc = (uint8_t *)kmallocc(sizeof(uint8_t) * TEST_KMALLOC_SZ);
20     while (1) {
21         if (show_stat) {
22             xil_printf("cfs_worker1 is running....\n");
23         }
24
25         if (pc != NULL) {
26             for (i = 0; i < TEST_KMALLOC_SZ; i++) {
27                 t = (uint8_t)(i % 256);
28                 pc[i] = t;
29             }
30         }
31     }
32
33     return 0;
34 }

```

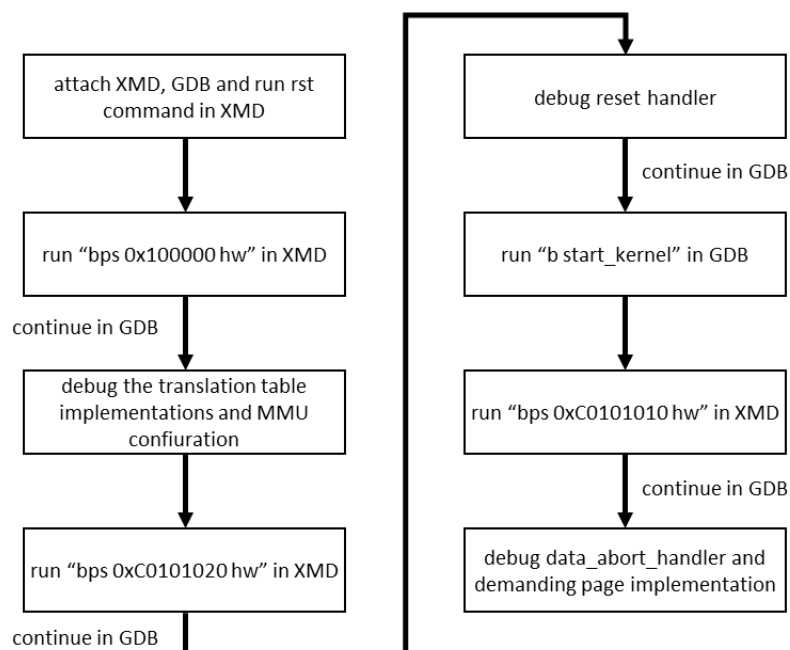
In line 9, cfs\_worker1 task allocate a 4KB memory in the virtual memory pool. Since the virtual memory pool is a lazy allocator, this adds the region descriptor in the meta data of the virtual memory pool. The first execution of line 12 makes an access to the address that is allocated in line 9 and makes a translation fault. This fault will follow the steps in figure ??? and allocate one entry in the first translation table and 1 entry in the second translation table and 1KB page frame in the physical heap memory. Since the first level entry covers 1MB memory region, the access to the next page address will generates the page fault.

Run git pull the latest source code and check out the virtual memory management by running “git checkout v5”. ‘v5’ is a tag for virtual memory management. Then you can build the source and follow the steps in chapter 3.7.3 to boot up the SLOS. Then attach the XMD debugger and GDB debugger as described

in chapter 3.8.1. After attaching the debuggers, run the “rst” command in XMD terminal. Now the processor is reset and it is in 0x0000\_0000. Connect the GDB debugger now and load the correct kernel symbols. To check the secondary bootloader, add a break point at 0x0010\_0000. This is the entry of kernel.elf and also the start of page translation table initialization. Run a “bps 0x100000 hw” command in XMD terminal and run “c” command in GDB terminal to have the processor continue to break point. When the processor is stopped at 0x0010\_0000, you can start your debugging with GDB debugger. Run a “disassem” command in GDB, then you can see the memory address and its assembler code. You can run a “si” command to run each assembler code. Then add another break point at init\_pgt() function in GDB debugger and continue to debug the translation table generation part. After return from init\_pgt() the ssbl code will enable the MMU. If the MMU is enabled, the address space used in the kernel is changed into the virtual address space. Let’s add another break point at 0xC0101020 which is the start address of reset\_handler of SLOS kernel. This break point should be added in XMD debugger by using “bps 0xC0101020 hw”. If you continue the processor in GDB, the processor should break at the reset\_handler. Now you can go back to the GDB and add a break point in start\_kernel(). You can go through the start\_kernel() code and also all the previous process management code now.

To see the demanding page, add a break point at 0xC0101010 which is the address of data abort exception handler. You should add this break point at XMD debugger. If you continue, the break point at data abort handler will be hit because the kmallocc() used in forkyl() will make a translation fault. The first kamllocc() should generate a section translation fault but from the second kmallocc() there is only page translation fault. This is because the first section translation fault will add one entry in the first level translation table and this covers 1MB memory region. You can add break points at handle\_fault() to see the demanding page implementation. If you add cfs\_work1() task, this will also generate a page fault and activate the page demanding.

Below figure??? describes the steps used to see the memory management implementation.



## References

- [1] [http://infocenter.arm.com/help/topic/com.arm.doc.ddi0035a/DDI0035A\\_7100\\_prelim\\_ds.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.ddi0035a/DDI0035A_7100_prelim_ds.pdf)
- [2] <http://www.ethernut.de/en/documents/arm-inline-asm.html>

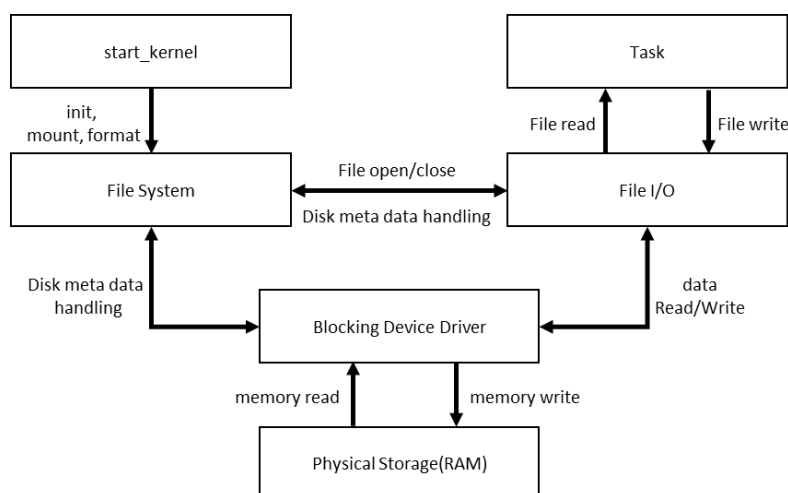
## 6 Storage Management

### 6.1 SLFS System

SLFS (Simple & Light File System) is a file system for SLOS.

#### 6.1.1 Design Architecture

SLFS high level work flow looks like below. The start\_kernel function initializes, mount and format the SLFS. After mounted, the SLFS can create a new file to access to the storage. SLOS has only ram disk storage and the access to this is simply memory read/write, which is a blocking access. Normal access to storage media such as HDD could be either blocking or non-blocking access. In blocking access, the task requesting read/write goes to sleep and wait until its read/write is done. In non-blocking access, the read/write to storage returns right after queuing the request. Since SLFS's read/write is memory load/store, which means the task is waiting but not sleep until the read/write is finished. The file system keeps the meta data of SLFS such as inode, data block allocation. These meta data is used through the file open/close, read/write access.



#### 6.1.2 SLFS Source Tree

To see the code implementations for SLFS, pull the sources from github and checkout v6.1 tag. This is already described in the chapter 3. Following files are added or modified for SLFS implementation.

File Path			Description
kernel/	core/	file_system.c	SLFS meta data block (super section, inode table, data block etc.) implementation.
		file.c	SLFS file manipulation implementation. file open, close, read, write implementation.

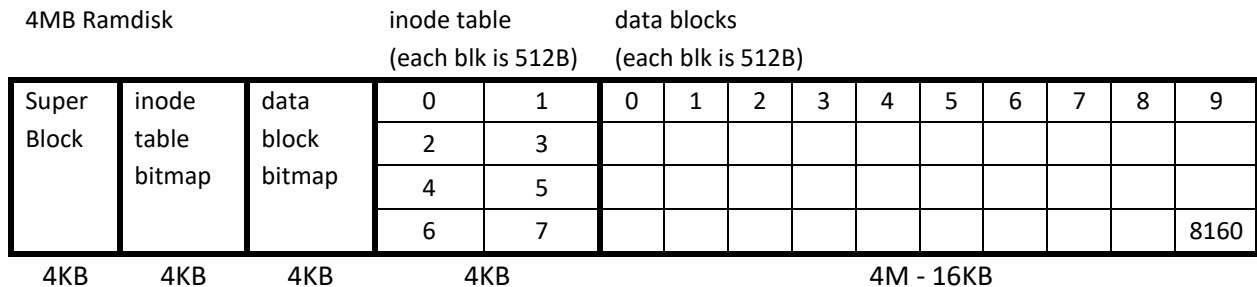


	ramdisk_io.c	ramdisk access implementation. This is a device driver of disk.
inc/	file_system.h	header files added
	file.h	header files added
	ramdisk_io.h	header files added

## 6.2 File System

### 6.2.1 Description of SLFS

SLFS file system is composed of super section, inode bitmap, data block bitmap, inode table and data block. The layout of ramdisk looks like below figure.



4KB Super Block in SLFS doesn't have any meaningful information. inode table bitmap and data block bitmap is a bitmap for the allocation of inode table and data block. One bit of the bitmap represents the occupation of a block in the inode table or in data blocks. 4KB inode table bitmap can cover  $4 * 1024$  entries, but SLFS just support only one 4KB size inode table which has only 8 inode entries. Since each block size in a ramdisk is 512B, 4KB data block bitmap can cover up to 16MB ( $4 * 1024 * 8 * 512B$ ) data block. Data block is an array of 512byte blocks that contains the data of a file.

### 6.2.2 inode for SLFS

inode is a meta data of a file. It contains all information that a file needs. inode structure in SLFS is as below.

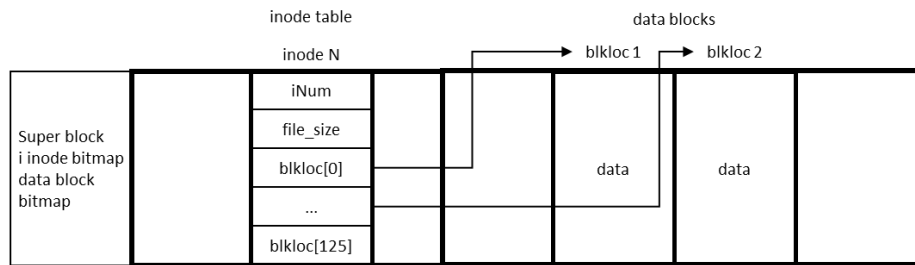
```

1 struct inode {
2     uint32_t iNum;
3     uint32_t file_size;
4     uint32_t blkloc[INODEBLKMAX];
5 };

```

As you can see, inode in SLFS is very simple. It has inode number, file size and index array of data block location. The block location index is the block number where the file data is located in the data block

region. That's all information that inode in SLFS has. The relation between inode blkloc index and data block location is depicted as below figure???



### 6.2.3 Mount and Format SLFS

SLFS is also needed to be mounted and formatted before being used. Mount is simply initializing the parameters of file system structure described in figure ????. Mount function looks like below.

```

1 void mount_file_system(void)
2 {
3     pfs->SuperBlkStartBlk = SUPER_BLK_START_BLK;
4     pfs->inodeBmpStartBlk = INODE_BITMAP_START_BLK;
5     pfs->DataBmpStartBlk = DATA_BLK_BITMAP_START_BLK;
6     pfs->inodeTableStartBlk = INODE_TABLE_START_BLK;
7     pfs->DataBlkStartBlk = DATA_BLK_START_BLK;
8
9     pfs->plBmp = (unsigned char *)(RAMDISK_START + INODE_BITMAP_START);
10    pfs->pDBmp = (unsigned char *)(RAMDISK_START + DATA_BLK_BITMAP_START);
11
12    pfs->bMounted = 1;
13 }

```

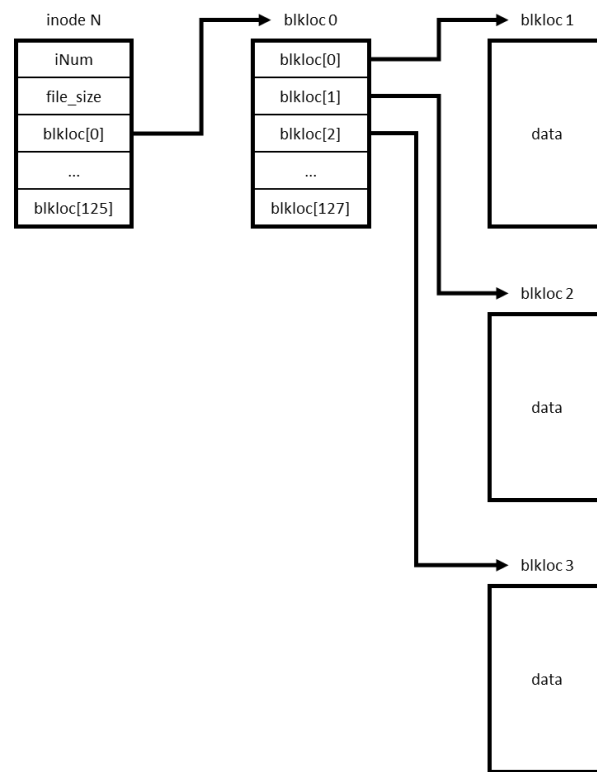
pfs is a pointer to file system structure variable that is allocated in the heap. The pfs is used to access each region in the file system. As you see, the mount\_file\_system() function initializes the SLFS structure with predefined offset values.

Format SLFS is wiping out all the data in ram disk. This erases the Super section, inode bitmap, data block bitmap, inode table and all data contents in the data blocks.

### 6.2.4 2 Level Data Block Indexing

inode in SLSF file system has an index array to point its data blocks. Since inode size is 512bytes, this array index size is limited to 504(512 - 8) byte that has 126 index array entries. This means the maximum data size that one file can have is 63KB (126 \* 512 byte). This looks too small for a file size. To increase

this max data size, we should increase the size of data block index array in inode. We can accomplish this simply by adding one more layer between this index array and the data block. The first data block indexed by inode blkloc array doesn't have file data. Instead, it has another block location index which has the file data. This 2 level data block indexing looks like figure ???



By adding this layer between data block and inode blkloc array, the max data size of one file can have is increased up to about 8MB (126 \* 128 \* 512byte). This is big enough for SLFS which has only 4MB ram disk.

## 6.3 File in SLFS

File in SLFS is nothing but an inode in the inode table.

### 6.3.1 Ramdisk I/O

File read/write needs a device driver functions that physically read/write to the storage device. SLFS has only ram disk for its storage, the device driver for disk I/O is just a memory read/write. This is a load/store operation in CPU perspective. Nonetheless, SLFS has a 512 byte of data block size, the read/write is done by this size. In other words, even writing 1 byte of data to a file occupies 1 block of data. Ram disk I/O functions are implemented by memory load/store with the size of 512 byte as below.

```
1 void read_ramdisk(int mem_blk_num, char *buf)
```

```

2      {
3          int i;
4
5          for (i = 0; i < DATA_BLK_SIZE; i++) {
6              buf[i] = ((char *) (RAMDISK_START + mem_blk_num * DATA_BLK_SIZE))[i];
7          }
8      }
9
10     void write_ramdisk(int mem_blk_num, char *buf)
11     {
12         int i;
13
14         for (i = 0; i < DATA_BLK_SIZE; i++) {
15             ((char *) (RAMDISK_START + mem_blk_num * DATA_BLK_SIZE))[i] = buf[i];
16         }
17     }

```

Line 5~7, 14~15 show that wrting/reading is simply done by memory load/store. Notice that the read/write operation is done with the size of 512 bytes. If we use a HDD or other, the device driver for those media should be implemented, but SLFS supports only ram disk to load user applications. SLOS allocate 4MB ramdisk into 0x300\_0000.

### 6.3.2 File Open, File Close and File Delete

File open should have a string parameter for the file name. This file name should be unique. Since there is no directory (or path to that file) in SLFS, the file name is the only way to discriminate the files. While opening a file, if there is already a file having the same name, the file open function will return a file pointer to that file. If there is not a file having the same name, then it will go through the steps to create a new file and return the file pointer of that. Below is the function declaration of file open in SLFS.

- struct file \*open\_file(char \*str)  
 str: a pointer to a string of file name.  
 return value: a file pointer to the file descriptor if there is a file with the same name as pointed by str. Otherwise, return a file pointer to a new file descriptor.

Opening a file in SLFS is nothing more than register a new inode to the inode table. File open allocates a file descriptor from heap memory and register an inode for the new file to the inode table. While doing that, it does some initialization of file data structure such as file size, file offset position pointer and increase the file open counter by one. After inode is registered, all R/W for that file is done via this inode.

Closing a file in SLFS decreases the opening counter of that file. Since SLFS doesn't have a directory, closing a file doesn't do any meaningful operations. Instead, in order to remove the file from the storage, there is a delete\_file() function. This function will unregister the inode, release data block allocated to that file and free the file descriptor of that file. So, the SLFS keeps the files in the ram disk after those files

are closed and have other applications access that file with the file name again. Below is the declaration of file close and file delete functions.

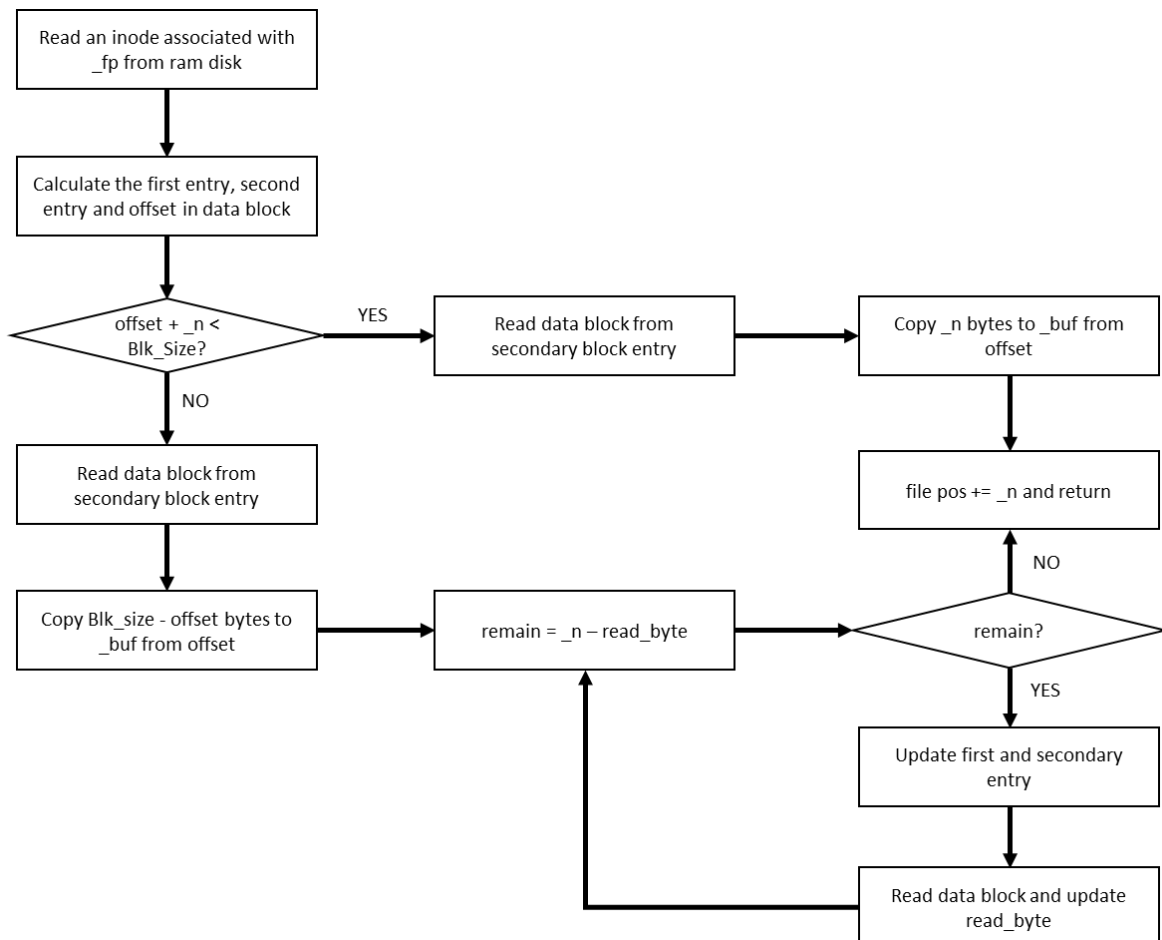
- `uint32_t close_file(struct file *fp)`
  - `uint32_t delete_file(struct file *fp)`
- fp: a file pointer  
return value: 0 if it is successful, positive number if it fails

### 6.3.3 File Read

File read function accesses to the data block by using inode. inode has the blkloc array which has the index of the data block allocated to that file. The read function's declaration looks like below.

- `uint32_t read(struct file *fp, uint32_t _n, char *_buf)`
- fp: a file pointer for reading  
\_n: data size to read  
\_buf: a pointer to a buffer to which data is copied  
return value: the data amount read

File read follows below flow. The implementation is in `kernel/core/file.c`



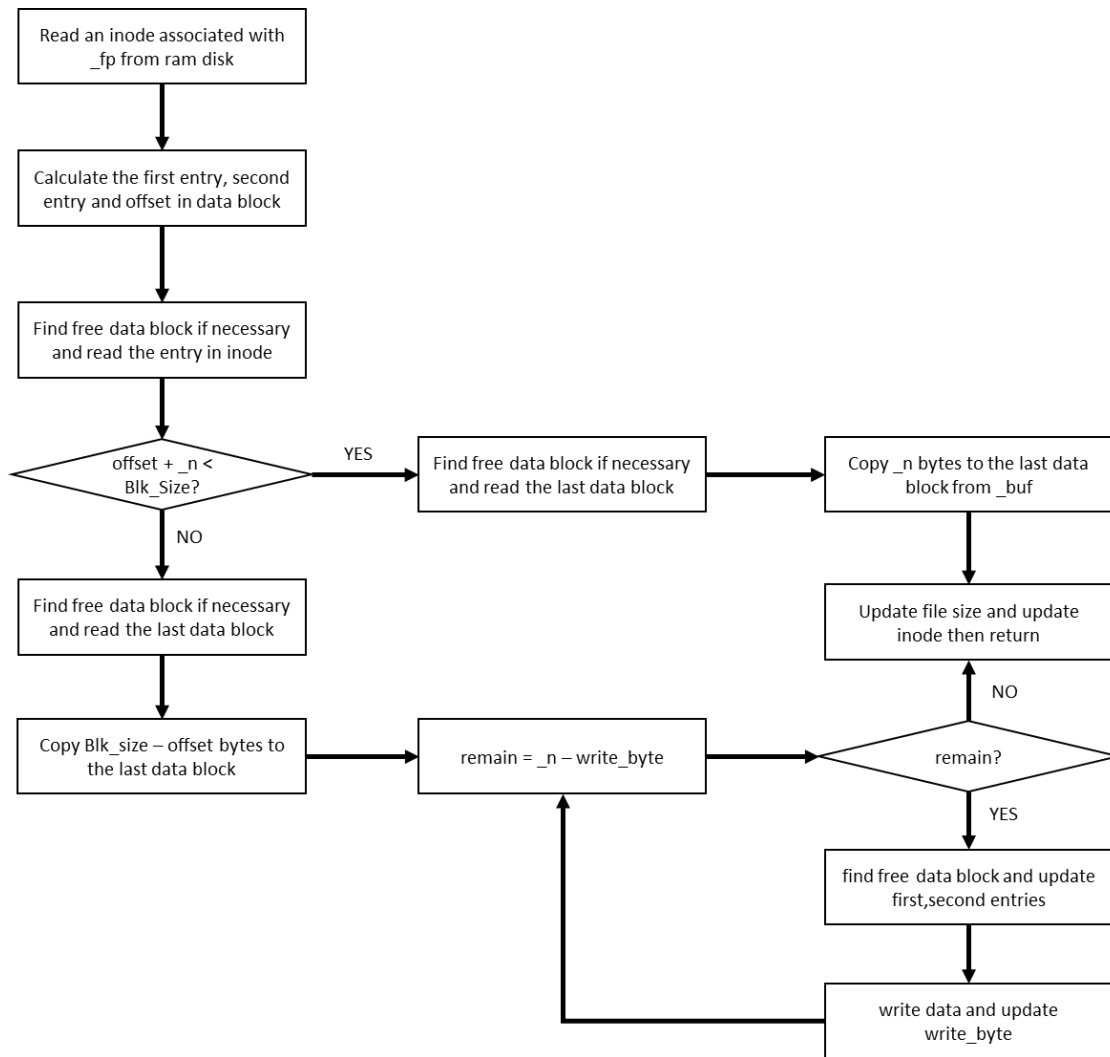
File read function first calculate the current file offset and entries of the first and second level table. Then, since the ram disk read happens only by 512 bytes, it checks whether current file read can be done within this block or not. If it is possible to read with current block, read function traverses the entries and reads the data block to out buffer. If it is not possible, then read function repeat that until the remaining bytes becomes zero. Finally, the read function updates its file position and return.

#### 6.3.4 File Write

File write function uses inode to access data block region. Unlike read function, write function needs to find out free blocks and to update the data block bitmap. Others are similar with the read function. Below is the write function declaration.

- `uint32_t write(struct file *fp, uint32_t _n, char *_buf)`
  - `fp`: a file pointer for reading
  - `_n`: data size to write
  - `_buf`: a pointer to a buffer from which data is copied
  - return value: the data amount written

File write follows below flow. The implementation is in `kernel/core/file.c`



The write function need to find a free data block. The `find_datablk()` function in file system does this. It traverses the data block bitmap and find a free data block then return the index of free data block. Then, the write function follows similar steps with read function; find a new data block and write until remaining data amount becomes zero.

### 6.3.5 File Read/Write Test

To test the file system implementation, let's run a task which does open a new file, write a predefined data to the file, read that file again and verify the read data with the original data.

We first need to add the initialization of the file system into the `start_kernel`. After `start_kernel()` function first initialize the memory management, initialize gic controller and idle task, it does the initialization routine for file system. The `start_kernel` is now changed as follows.

```

1  int start_kernel(void)
2  {

```

```

3      struct framepool framepool;
4      struct pagetable pgt;
5      struct vmppool kheap;
6
7      init_kernmem(&framepool, &pgt, &kheap);
8      init_gic();
9      init_idletask();
10
11     init_file_system();
12     mount_file_system();
13     format_file_system();
14
15     init_rq();
16     init_wq();
17     init_shell();
18     init_timertree();
19     init_cfs_scheduler();
20     init_timer();
21     update_csd();
22     timer_enable();
23     cpuidle();
24
25     return 0;
26 }

```

Line 11 ~ 13 are added for SLFS. These lines are for initialization of SLFS, mount and format the file system.

Now that we have a file system, we can try to create a new file and to write, read with the file descriptor. Following is a routine to test the file read/write routine.

```

1      fp = open_file("test");
2      for (i = 0; i < FILE_TEST_LEN; i++)
3          buf[i] = i % 256;
4      write(fp, FILE_TEST_LEN, buf);
5      reset(fp);
6      read(fp, FILE_TEST_LEN, temp);
7      xil_printf("file test : ");
8      for (i = 0; i < FILE_TEST_LEN; i++) {
9          if (buf[i] != temp[i]) {
10             xil_printf("fail!!\n");
11             break;
12         }

```



```

13     }
14     if (i == FILE_TEST_LEN) {
15         xil_printf("pass!!\n");
16     } else {
17         xil_printf("fail!! i: %d\n", i);
18     }
19
20     close_file(fp);
21     fp = NULL;

```

Line 1 opens a new file with the name “test”. This file name should be unique for each file. SLFS checks whether that file already exists or. If it doesn’t exist, SLFS creates a new file then return a file pointer of that file.

Line 2 ~ 4 writes a character data to the file. Be careful the buf array size is not too big if it is declared as a local variable. Since every task in SLOS has 4KB stack size, if this buf array is too big, there could be a stack overflow. If you want to use big buffer to write a file, then allocate this array in the heap by using kmalloc.

Line 5 reset the file position to the first location for the next reading operation.

In line 6, file read operation saves the read data to temp buffer.

The rest line 8 ~ 18 verifies the file read / write operations. If they work properly, then this will print “pass” string to the serial terminal. If they don’t, “fail” string with failed position is printed.

Then line 20 ~ 21 close that file. Even after close this file, that file remains in the file system. You can still access that file with open\_file() afterwards. The delete\_file() function will remove the file permanently.

## 6.4 System Call

User mode applications have no idea about the system hardware. The applications don’t know how to allocate heap memory for its use and don’t know how to save its data to HDD. System call allows user application to access those hardware resources through the kernel.

SLOS has a ‘write’ system call to export the ability to write characters in the buffer to UART. With this system call, an application can print a string message to the terminal window.

There is an ARM command for system call. The ‘SVC’(Supervisor call) or formerly ‘SWI’(Software Interrupt) command is used. These commands are used to request privileged operations or access to system resources from operating system.

Each system call has a unique number. This number is used to indicate what the caller is requesting. The write system call of SLOS has decimal 2.

### 6.4.1 SVC Handler Implementation

The libslos/syscall.S file has the implementation of system call. The system call implementation for write is as below.

```

1  write:
2      mov r12, lr
3      svc #2
4      mov pc, r12

```

Calling SVC command in line 3 is all for system call implementation. The SVC command in line 3 embeds the decimal number 2 to its command. 32bit ARM SVC command is encoded as below.



The imm24 has the value of system call number. The kernel system call handler parses this command and extract the system call number from imm24 field.

When the SVC is run, the ARM core goes to syscall exception handler which has the offset 0x8 from the exception vector base address(VBAR). Since SLOS has 0xC0101000 as its VBAR, the syscall exception handler address is 0xC0101008. Below is the implementation of syscall handler.

```

1  syscall_handler:
2      msr    cpsr_c, #MODE_SVC | I_BIT | F_BIT
3      stmfd  sp!, {r0-r12,lr}
4      ldr    r12, [lr,#-4]
5      bic    r12, #0xff000000
6      mov    r2, r12
7      bl     platform_syscall_handler
8      mrs    r0, CPSR
9      bic    r1, r0, #I_BIT|F_BIT
10     msr    cpsr_c, r1
11     ldmfd  sp!, {r0-r12,pc}

```

Line 4 loads the SVC command and line 5 ~6 is masking out the upper 8bits and put the system call number into r2. The platform\_syscall\_handler declaration is below.

```
int platform_syscall_handler(char *msg, int idx, int sys_num)
```

The first parameter is a buffer pointer from user application. The string buffer can be associated with this pointer parameter. The second parameter is the user task index. SLOS kernel can calculate the user task's starting address with this index. The third parameter of platform\_syscall\_handler is the system call number. Below is the platform\_syscall handler implementation.

```

1  int platform_syscall_handler(char *msg, int idx, int sys_num)
2  {

```

```

3      int ret = 0;
4
5      switch (sys_num) {
6          case SYS_WRITE:
7              msg = msg + (USER_APP_BASE + USER_APP_GAP * idx);
8              xil_printf(msg);
9              break;
10         default:
11             break;
12     }
13     return ret;
14 }

```

SYS\_WRITE system call gets the address of message buffer and prints it out through xil\_printf() function that is sending the character string to the terminal. xil\_printf() function accesses the UART registers, which needs a privileged permission. The user application can access the system hardware by using the system call(SVC) command like this way.

#### 6.4.2 Syscall Library for User Application

To expose the system call to user application which is built as a different binary, there is a libsls.a library. This library is composed of two files; print\_mesg.c and syscall.S in libsls directory.

print\_mesg.c has the function of print\_mesg() which is supposed to be used by user application. The print\_mesg function implementation is below.

```

1  int print_mesg(const char *buf, const int idx)
2  {
3      write(buf, idx);
4      return 0;
5  }

```

The print\_mesg() function is very simple. It calls only the write() function which is in syscall.S. write() function is covered in chapter 6.4.1. It just runs a supervisor call(SVC).

syscall.S has the implementaton of system call. Currently, there is only one system call which is for printing message to the terminal through UART system hardware. Other system calls also can be added to this file.

libsls.a can be built by arm-none-eabi-ar command. This command archives the compiled object files into a single file and used to create a library holding commonly used subroutines. To build the libsls.a, add below changes to the Makefile.

```

1  $(OUT_TOP)/libsls/libsls.a : $(OUT_TOP)/libsls/syscall.o $(OUT_TOP)/libsls/print_mesg.o
2      $(AR) rc $@ $^

```

This command in Makefile will archive the syscall.o and print\_mesg.o object files to libslos.a library. The user application can link this library statically to use the system calls.

## 6.5 HelloWorld Application and Ramdisk

ramdisk.img is built for building user applications.

### 6.5.1 HelloWorld Application

The helloworld user application prints a string message into the terminal window. This application accesses the system hardware(UART) via the system call that is exposed by the libslos.a library.

```
1 void main(void)
2 {
3     const char *a = "hello world!!\n";
4     print_mesg(a, 0);
5     while (1);
6 }
```

print\_mesg() function is implemented in libslos.a. Since there is no exit() system call in SLOS, the user application should loop forever at the end of its flow. Since there is no runtime linker in SLOS, the object file of helloworld application should be statically linked with the libslos.a in below Makefile.

```
1 $(APPS) : apps/helloworld.c
2     $(CC) -o $(OUT_TOP)/ramdisk/helloworld.o -c $< -g
3     $(LD) -o $(APPS) $(OUT_TOP)/ramdisk/helloworld.o -e main -L$(OUT_TOP)/libslos -lslos -static
```

The -static option in line 3 means the output helloworld binary is statically linked with libslos.a library. With this, the helloworld binary can print the message to the terminal with system call.

### 6.5.2 Building ramdisk.img

ramdisk.img is a file containing all user applications. It is built by mkfs executable. mkfs is a standalone linux application and its source is in mkfs/mkfs.cpp. mkfs reads the user application and appends it to the end of ramdisk.img. The first 4 bytes in ramdisk.img is the number of applications embedded into ramdisk.img. Next, there is 4 bytes of size of the user application. The user application must be padded to make 4 byte alignment. If there is padded bytes, the size is updated with the padding bytes before written into the ramdisk.img. Then, the user application binary code which is either padded or not is appended. The next user application is concatenated to the end of the file in the same way. The final ramdisk.img is built simply as below.

app. Number	app #1 size with padding	app #1 binary	....	app #N size with padding	app #N binary
-------------	-----------------------------	---------------	------	-----------------------------	---------------

### 6.5.3 Loading ramdisk.img from fsbl

The ramdisk.img is a separate file from kernel.elf. Normally, the final OS image is built by merging the kernel image and init ramdisk image and the bootloader loads these two binaries into corresponding address. But the bootloader for SLOS loads only the kernel.elf. So, we are going to tweak a current Xilinx bootloader(zynq\_fsbl.elf) to load the ramdisk.img to a specific memory location. After you open the zynq\_fsbl project which was created in chapter 2.6.2, go to zynq\_fsbl/src/ and open fsbl\_hook.c. Then, search FsblHookBeforeHandoff function. This function is called in the fabi just before jumping to the kernel.elf. You can't see any meaningful implementations there. Modify that function as below.

```

1  #include "ff.h"
2  #include "sd.h"
3  #define TEMP_SCRATCH_ADDR 0x20000000
4  u32 FsblHookBeforeHandoff(void)
5  {
6      char buffer[32];
7      UINT br;
8      u32 Status;
9      u32 LengthBytes;
10     u32 ImgCnt, i;
11     FRESULT rc;
12     FIL fp;
13     char *boot_file;
14     char *pScratch;
15     Status = XST_SUCCESS;
16     /*
17      * User logic to be added here.
18      * Errors to be stored in the status variable and returned
19      */
20     strcpy_rom(buffer, "ramdisk.img");
21     boot_file = (char *)buffer;
22     rc = f_open(&fp, boot_file, FA_READ);
23     if (rc) {
24         fsbl_printf(DEBUG_GENERAL, "SD: Unable to open file %s: %d\n", boot_file, rc);
25         return XST_SUCCESS;
26     }
27     rc = f_lseek(&fp, 0x0);
28     if (rc) {
29         return XST_SUCCESS;

```

```

30     }
31     pScratch = (char *)TEMP_SCRATCH_ADDR;
32     f_read(&fp, pScratch, 4, &br);
33     ImgCnt = (u32)((u32 *)(pScratch));
34     pScratch += 4;
35
36     for (i = 0; i < ImgCnt; i++) {
37         f_read(&fp, (void*)(pScratch), 4, &br);
38         LengthBytes = (u32)((u32 *)(pScratch));
39         pScratch += 4;
40         rc = f_read(&fp, (void*)(pScratch), LengthBytes, &br);
41         if (rc) {
42             f_close(&fp);
43             return XST_SUCCESS;
44         }
45         pScratch += LengthBytes;
46     }
47
48     if (rc) {
49         fsbl_printf(DEBUG_GENERAL, "*** ERROR: f_read returned %d\r\n", rc);
50     }
51     f_close(&fp);
52     fsbl_printf(DEBUG_INFO, "In FsblHookBeforeHandoff function \r\n");
53
54     return (Status);
55 }

```

This function reads the ramdisk.img from sd card and load it to the TEMP\_SCRATCH\_ADDR. The TEMP\_SCRATCH\_ADDR is defined as 0x200\_0000. This function is also using the zynq\_fsbl\_bsp's file library functions such as f\_open, f\_read and so on. We are not interested in those implementations. We just want to load the ramdisk.img file to a specific location. This hook function does that. Even if you don't want to build this modified bootloader by yourself, you can download the prebuilt bootloader from the tools repository. Run below command to clone the tools.

```
git clone https://github.com/chungae9ri/tools
```

Then, you can see the zynq\_fsbl\_hook.elf. This file is a modified bootloader and include this to build the BOOT.bin.

## 6.6 ELF Loader for User Application

Let's load helloworld application and run it.

### 6.6.1 Creation Application Files

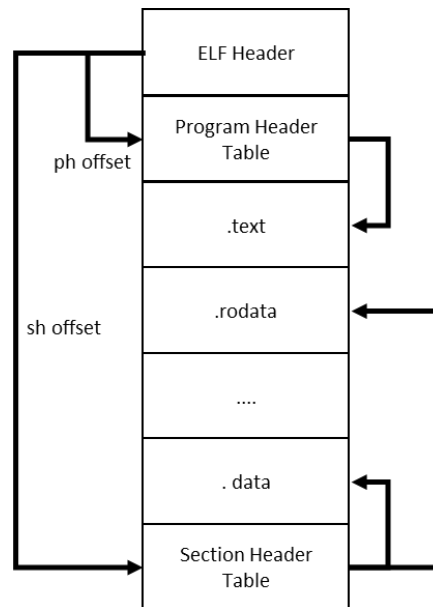
At the end of `start_kernel`, the `create_ramdisk_fs()` function accesses user application executables in the ramdisk image and create application files in the SLFS. This ramdisk image is already loaded by bootloader into `TEMP_SCRATCH_ADDR`. Creation of user application files is accomplished by file write of the loaded ramdisk image. Since the ramdisk located at `TEMP_SCRATCH_ADDR` is same as the figure ??, we can parse it and write the binary data to a file. `create_ramdisk_fs()` looks like below.

```
1  int32_t create_ramdisk_fs(void)
2  {
3      /* variable declarations here */
4
5      offset = 0;
6      appCnt = *((uint32_t *)SCRATCH_BASE);
7      offset += 4;
8
9      for (i = 0; i < appCnt; i++) {
10         szApp = *((uint32_t *) (SCRATCH_BASE + offset));
11         offset += 4;
12         psrc = (char *) (SCRATCH_BASE + offset);
13         sprintf(fname, "App_%u", (unsigned int)i);
14         fp = open_file(fname);
15         if (fp) {
16             write(fp, szApp, psrc);
17             close_file(fp);
18             fp = NULL;
19         } else {
20             return 1;
21         }
22         offset += szApp;
23     }
24
25     return 0;
26 }
```

Line 6 reads the application count. The application count is the first 4 byte of `SCRATCH_BASE` address. Line 9~23 loops until the application count and writes the application binary data to the file in SLFS. When application file is created, the name is chosen as “App\_index”. This name is used to access that file later. After this, the helloworld executable file named “App\_0” is placed at the ram disk storage. If there are more user applications in the ramdisk image, the executable files name “App\_#N” are placed in ramdisk.

### 6.6.2 Elf

Elf(Executable and Linkable Format) is a common standard file format for executable files, object code, shared libraries and core dumps[1]. Elf is a container of binary code, string, symbols. Elf has headers to describe these contents. Elf file looks like below.



ELF file has composed of two components; section and segment.

- **Section**  
Section holds a bulk of object file information for linking: instructions, data, symbol table, relocation information and so on. A section header table has the entries of sections.
- **Segment**  
Segment holds the information how to create process image. Multiple sections can be integrated into one segment. A program header table has the entries of segments.

We can access all program segment by going through ELF header->Program header->Segment. Also, we can access section information by going through ELF Header->Section Header->Section.

### 6.6.3 Loading User Application ELF

We already created application executable files in SLFS in chapter 6.6.1. Now, we will load it to memory and run it. `load_ramdisk_app()` read the application file from SLFS, parsing the ELF headers, finding the entry point of the binary and run `forkyi` to run the application. `load_ramdisk_app()` function is:

```
1 void load_ramdisk_app(uint32_t appIdx)
2 {
3     /*
4      * do basic stuffs
5      */
```



```

6
7     app_load_addr = (char *)(USER_APP_BASE +
8                         APP_LOAD_OFFSET +
9                         USER_APP_GAP * appIdx);
10
11     sprintf(temp, "App_%u", (unsigned int)appIdx);
12     fp = find_file_by_name(temp);
13     if (fp) {
14         read(fp, fp->fsz, app_load_addr);
15         load_elf(app_load_addr, appIdx);
16     }
17 }

```

Line 7~9 sets the application loading address which is 0x180\_0000. Line 12 finds the application file named App\_0 which is for our 'helloworld' application created in chapter 6.5.1. If it is successful, line 14 read it from SLFS to the loading address 0x180\_0000. Next, the line 15 parses this ELF file and run the executable. The steps of running the user executable is finding the entry point and calling the forkyl() function with that entry point. Below is the load\_elf function implementation.

```

1     task_entry load_elf(char *elf_start, uint32_t idx)
2     {
3         /*
4          * variable declaration and initialization here
5          */
6
7         hdr = (Elf32_Ehdr *) elf_start;
8
9         exec = (char *)(USER_APP_BASE + idx * USER_APP_GAP);
10        phdr = (Elf32_Phdr *) (elf_start + hdr->e_phoff);
11
12        for (i = 0; i < hdr->e_phnum; ++i) {
13            if (phdr[i].p_type != PT_LOAD) {
14                continue;
15            }
16            if (phdr[i].p_filesz > phdr[i].p_memsz) {
17                xil_printf("load_elf:: p_filesz > p_memsz\n");
18                return 0;
19            }
20            if (!phdr[i].p_filesz) {
21                continue;
22            }
23

```

```

24         start = elf_start + phdr[i].p_offset;
25         taddr = phdr[i].p_vaddr + exec;
26         for (j = 0; j < phdr[i].p_filesz; j++)
27             taddr[j] = start[j];
28     }
29     shdr = (Elf32_Shdr*)(elf_start + hdr->e_shoff);
30
31     for (i = 0; i < hdr->e_shnum; i++) {
32         if (shdr[i].sh_type == SHT_SYMTAB) {
33             strings = elf_start + shdr[shdr[i].sh_link].sh_offset;
34             entry = (task_entry)find_sym("main", shdr + i, strings, elf_start, exec);
35             break;
36         }
37     }
38
39     sprintf(buff,"user_%u",(unsigned int)idx);
40     create_usr_cfs_task(buff, (task_entry)entry, 4, idx);
41
42     return entry;
43 }

```

This function is composed of two parts. First one is finding the loadable segment in the program headers and copies that segment into the execution memory region. The other part is finding the 'main' entry address by browsing the symbol table.

Line 12 ~28 loads the loadable segment to the memory. This routine goes through all the program header entries and if the type of the segment is PT\_LOAD, then it copies that segment to the execution memory which is starting from 0x100\_0000. When copying the segment to the execution memory, the segment virtual address(p\_vaddr of the segment) must be added to the base address. Now, we loaded the execution segment to the memory. Next step is finding the entry point of the runnable segment. Finding the entry point is finding the 'main' function address. This can be done through the symbol table of section header. Line 31 ~ 37 is the routine finding the address of main entry point. For this, it first find the symbol table section. The section header type(sh\_type) of symbol table is defined as SHT\_SYMTAB which is the decimal value 2. Then, we can get the string table section address by using the symbol table section header's sh\_link which is the section header index of associated string table. The line 33 does this. Since we have the string table, next step is traversing the string table and looking for the 'main' string.

String table sections hold null-terminated character sequences, commonly called strings. The object file uses these strings to represent symbol and section names. One references a string as an index into the string table section. The first byte, which is index zero, is defined to hold a null character. Likewise, a string table's last byte is defined to hold a null character, ensuring null termination for all strings. A string whose index is zero specifies either no name or a null name, depending on the context. An empty string table section is permitted; its section header's sh\_size member would contain zero. The section header name field (sh\_name field) in section header and the symbol name field(st\_name) in symbol table entry hold an

index of this string table. The following figure shows a string table with 25 bytes and the strings associated with various indexes.

Index	0	1	2	3	4	5	6	7	8	9
0	\0	n	a	m	e	.	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

The string table entries index is:

Index	String
0	none
1	name.
7	Variable
11	able
16	able
24	null string

As the example shows, a string table index may refer to any byte in the section.

By using this string table and symbol table, find\_sym function ins line 34 find the entry point of helloworld application. The find\_sym() function is:

```

1  Elf32_Addr find_sym(const char* name,
2      Elf32_Shdr* shdr,
3      const char* strings,
4      const char* src,
5      char* dst)
6  {
7      int i;
8
9      Elf32_Sym* syms = (Elf32_Sym*)(src + shdr->sh_offset);
10     for (i = 0; i < shdr->sh_size / sizeof(Elf32_Sym); i++) {
11         if (strcmp(name, strings + syms[i].st_name) == 0) {
12             return (Elf32_Addr)(dst + syms[i].st_value);
13         }
14     }
15 
```

```

16         return -1;
17     }

```

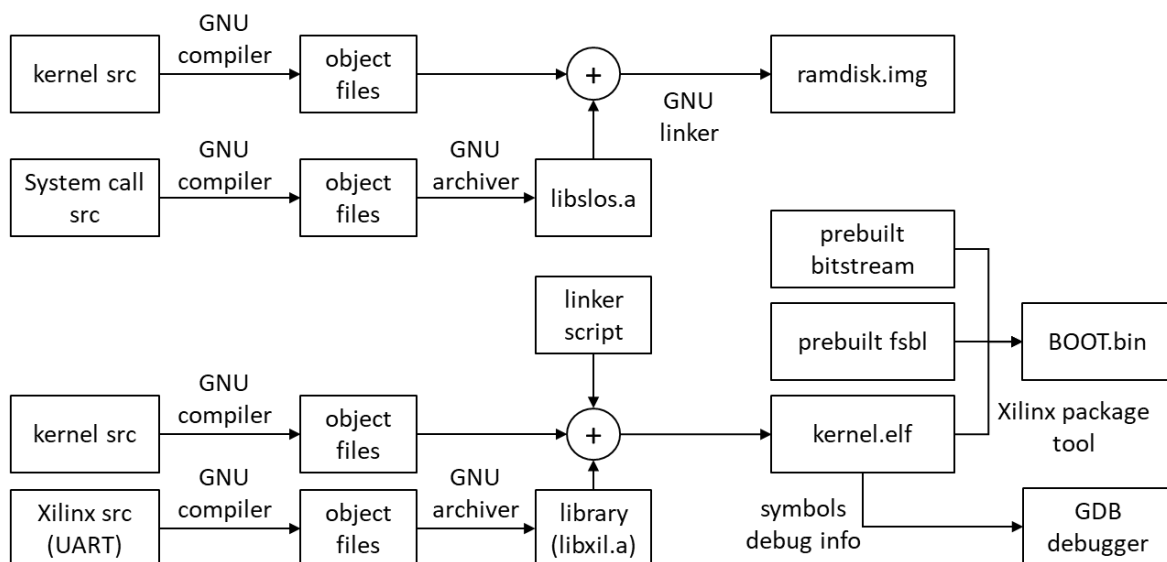
As mentioned before, the `st_name` field in the symbol table entry has the index of string table. The loop in line 10 traverses the symbol table entries and compare the symbol's string with 'main' string to find the 'main' symbol's entry. If succeed in finding the 'main' symbols entry in the symbol table, it returns the `st_value` which is the virtual address of 'main' symbol.

After finding the entry point address, the `create_usr_cfs_task()` function in the previous source fork a new task with that entry. This creates a new CFS task. Now the helloworld application is scheduled by the CFS scheduler with other tasks.

## 6.7 Putting It Altogether

### 6.7.1 Changes in Build Process

Let's load helloworld application and print a message through the system call. The build process in chapter 3.7.2 has a little modification because of `ramdisk.img`.



Now, we need to copy 2 images to SD card; `ramdisk.img` and `BOOT.bin`. `ramdisk.img` file has the helloworld application. We also need to use the new prebuilt fsbl which loads the ramdisk image to `0x200_0000`. For these changes in the build process, there is a few modifications in the Makefile.

### 6.7.2 Let's see it

In the project repository, run following command:  
`git checkout v6.7`

Then, run make command in the slos directory. Copy the out/ramdisk/ramdisk.img into SD card and build the BOOT.bin with prebuilt FPGA bitstream and new fsbl. The new fsbl can be downloaded from tools repository. After copying those files, try to boot the Xilinx evaluation board with UART terminal connected. After hit the enter key, you can see the shell prompt. The shell task displays the command list. There is a new command: 'apprun'. Run that command and see whether we can see the "helloworld" message.

```
shell >
taskstat, whoami, hide whoami
cfs task, rt task, oneshot task
sleep, run
apprun
shell > apprun
shell > hello world!!
nice to meet you!!
I am user_0 app!!
```

The helloworld application prints messages through system hardware resources. This means the system call is working in SLOS. Now, run the 'taskstat' command to see the CFS task status.

```
shell > taskstat
cfs task:idle task
pid: 0
state: 0
priority: 16
jiffies_vruntime: 231
jiffies_consumed: 491

cfs task:shell
pid: 1
state: 0
priority: 2
jiffies_vruntime: 231
jiffies_consumed: 3927

cfs task:cfs_worker1
pid: 2
state: 0
priority: 8
jiffies_vruntime: 231
jiffies_consumed: 982

cfs task:cfs_worker2
pid: 3
state: 0
priority: 4
jiffies_vruntime: 232
jiffies_consumed: 1972

cfs task:user_0
pid: 4
state: 0
priority: 4
jiffies_vruntime: 232
jiffies_consumed: 1972
shell > █
```

As you can see, the virtual runtime is pretty similar among tasks which means those tasks are sharing the cpu time in quite fair way.

[1] [https://en.wikipedia.org/wiki/Executable\\_and\\_Linkable\\_Format](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format)

## **7 Design a Custom HW in FPGA and Device Driver**