A Little Book on the Custom OS Developement from Scratch

Third Edition

Kwangdo Yi

Copyright © 2021 by Kwangdo Yi

All rights reserved. No part of this book may be reproduced or used in any manner without written permission of the copyright owner except for the use of quotations in a book review. For more information, address: kwangdo.yi@gmail.com

Publisher: Kwangdo Yi
First Printing: 2021
ISBN: 978-1-71614-465-3
THIRD EDITION
www.abandoned-engineer.com

To my beloved better half, Jooyeon and
my little friends, Jinseo and Hyunseo

Contents

Preface

When I first started my career as a software engineer, I felt there are too many things that I don't know anything about. I was scared when my senior engineer asked me about technical issues. As time goes by, I realized that other engineers also don't know that much. Then, how the machines, computers, chipsets, smartphones built by those engineers are working? It didn't take much for me to figure out the secret. Curiosity!

After that, I craved for going deeper and deeper. In fact, the company I worked for at that time was a smartphone manufacturing company and mostly bought the solutions from other companies. Lead time is the priority in that kind of industry. Nonetheless, I still wanted to know how this works, why that doesn't work, but it was so hard to dig into those area in the company. They wanted to solve the problem quicker than their competitors and perfered closing those issues through the vendor company. At the time when I almost died for curiosity, I decided to be a code generator, not code consumer any more. So, I started to work on the development of custom operating system.

I had many hardware evaluation boards at that time and most of them were thrown away after project was completed. I decieded to reuse one of them. I tweaked some bootloader code and developed a very simple context switching code. I was in seventh heaven when I first saw its working. Nobody told me on how to do this, but I designed, found a way to make it work from what I had.

After that, I named this operating system an SLOS and added small pieces one by one. Since I had to do this after company work, the pace was very slow and sometimes I got stuck somewhere I couldn't solve. But as this self-motivated project progressed, I was sure that making a custom operating system is not that difficult and anybody can do this as well. Building a commercial maturity OS must be so hard and it is impossible to do that personally. But creating something for fun is still worth personally.

Recently, I felt that I stuck again with this fun project. Actually, this kind of work is an endless work. There are still many bugs, many unfilled features, and so on. I felt I need to wrap up this journey for my next step further. Then, what is the best way to do this? A wrap-up party? I thought the best way is writing a book on my journey and shares my experience with others. I encourage others who stay also hungry can do this kind of work as well. Most of engineers that I have met in my career are smarter than I, then why don't they do the better projects for their curiosity.

Here, I am sharing the record of my journey. While sailing this fun, fantastic journey, I must give a deep appreciation to my family who board in the same boat. Thank you so much Jooyeon, Jinseo and Hyunseo.

Preface for 3rd Edition

It has passed three years after publishing the first edition and I found many errors and unfilled features. I have been adding new features such as bitbake build, SMP SLOS etc., and fixed many bugs since then. I refreshed the old book and SLOS source code now and happily share it with others who are anxious to start their hobby project and with others who stay foolish and hugry.

I have to admit that there are still bugs and non-professional implementations, so please bear with me on this. But I am wide open to the feed back and welcom any comments.

1 What is SLOS?

1.1 Introduction

A major role of *operating system (OS)* is orchestrating system's hardware and software resources. It is placed in the middle of hardware and software. Figure 1-1 shows a layered model for a personal computer. Operating system itself is a software and located at the edge boundary of software, and just on top of hardware as in figure 1-1.

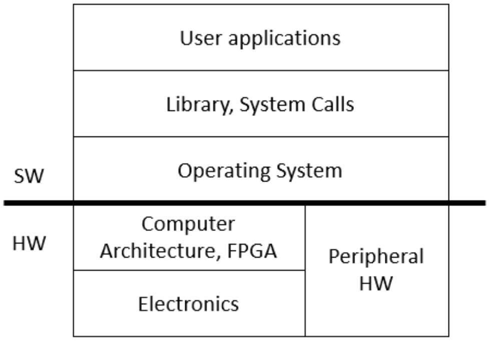


Figure 1-1: Layered model of a computer

Operating system loads user applications, allocates appropriate CPU time to the applications and at the same time, it manages computer's hardware resources, exposes the hardware's functionality to the higher layer's software applications. It is an interface channel between hardware and software. It includes device drivers to manages all kinds of peripheral hardwares. Examples of hardwares resources are CPU processor, memory, storage devices, peripheral I/O devices such as display, USB, UART and so on. Softwares includes user-developed applications, system calls, libraries and device drivers. Since there are different types of these hardware and software and their combinations, operating system has also variant types to accomplish its best work on different target systems. For example, operating system for *mainframe* is designed to support multi users and is optimized for high performance. Operating system for personal computer is developed to support business applications, internet applications, games, entertainment and so on. Operating system for mobile device such as smartphone is highly optimzed for power consumption and for network connectivities. Also, there are simple operating systems for embedded devices which have very limited hardware resources targeting a very specific need. Thus, some operating system is good at high performance jobs and some operating system is good at entertainment and business jobs, and some operating system is good for mobility and connectivity. Different operating system is designed and developed for different needs of work.

Operating System is a term for a bigger concept including *kernel,* device drivers, libraries, services and applications. It is kernel that is placed in the center of the operating system modules. The way how modern operating system implements its kernel to achieve these goals falls into one of below two big categories.

1) *Micro Kernel*

   Micro kernel is normally composed of separate kernel core, services and *IPC (Inter Process Call).* It means each sub module has its own execution binary. Kernel core is for core functionalities like *scheduling, memory management.* Most operating system jobs run as services such as file system, device drivers. The good thing of Micro Kernel is that these services run in *user mode* and failure in these services doesn't impact on the kernel core or other services.

   In Micro Kernel, the services and kernel core communicate each other through an IPC. Micro kernel block diagram looks like figure 1-2. Applications, services, device driver and even kernel itself work as separate entities and communicate each other through IPC in figure 1-2.

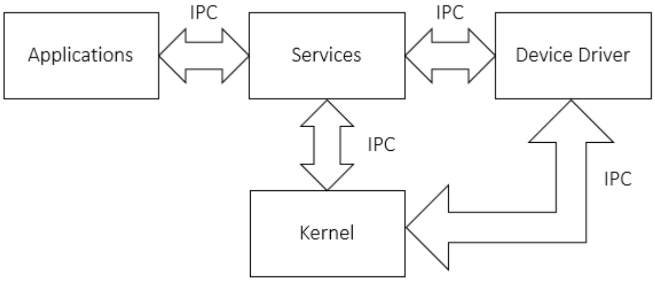


Figure 1-2: Micro Kernel Structure

   When I worked on Windows Mobile (This OS was developed by Microsoft and is quite old now and had gone away from the mobile industry long ago) product, the device driver was just a *DLL (Dynamic Link Library)* binary. A failure in this device driver doesn't impact the kernel operation. After debugging the problem in the device driver, and copying it into a correct location and relaunching the device driver are all necessary steps to test the new device driver. In Micro Kernel, since each sub-block (kernel, services, device driver) has its own memory space and works separately, the kernel is very stable. It isn't affected by a mal-formed 3rd-party device driver. The downside of Micro Kernel is that performance could be worse because there are more overheads in *context switching* and IPC calls.

2) *Monolithic Kernel*

Contrary to the multiple executable files in Micro Kernel, the Monolithic Kernel is one big executable file including all kernel subsystem (kernel core, device drivers not including the user applications and service daemons) in it. This is a big kernel that includes all kernel submodules in one executable binary and all these submodules share the same memory space. Its structure is as simple as Figure 1-3.

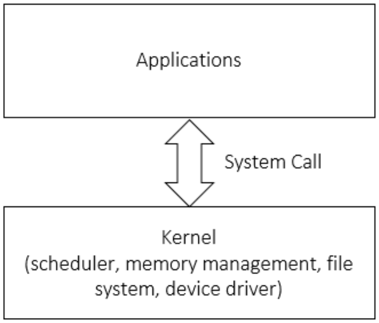


Figure 1-3: Monolithic Kernel Structure

The kernel core shares its memory space with other kernel subsystems. All subsystem modules can communicate each other without any special method like IPC. The user applications still can't touch the kernel area. They can get the operating system service only through the predefined *system calls*. Normally, user application and kernel take non-overlapping, its own memory address space in the whole memory space and work as a one big application together. For example, the lower 3GB address space is dedicated to user application including libraries, and the higher 1GB address space is assigned to the monolithic kernel usage. In this aggregated design, a defect in the custom device driver can generate the whole kernel crash or kernel panic.

A good example of Monolithic Kernel is the *Linux*. After building a Linux kernel, there is one *ELF (Executable and Linkable Format)* binary which contains all subsystems such as kernel core, device drivers, file system, memory management, network, platform architecture code and so on. A simple memory contamination from a custom device driver makes the kernel panic while the Linux kernel boots up.

So, the Monolithic Kernel is less stable than Micro Kernel. There is much possibility that user code corrupts the kernel memory space. Instead, it has a better performance compared to Micro Kernel because of lesser communication overhead.

SLOS used in this book to demonstrate a custom OS development is an acronym of *Simple and Light Operating System*. SLOS is monolithic kernel, which means all kernel submodules in SLOS are integrated into one ELF binary. Most of SLOS features belong to the embedded operating system category. Even though SLOS is designed for studying and is developed for fun, some high-level OS features are also covered as well. You might feel that SLOS features are not organized well compared to commercial operating system, or some features in SLOS are not necessary. But while reading this book, please keep in mind that SLOS is not a commercial maturity OS and all implementation is done for fun and personal curiosity on OS feature implementations. So, we can touch unnecessary OS features such as building custom hardware with FPGA if we have interests.

Before delving into our custom methods on developing a simple operating system, let's look into more on the features of current commercial operating systems. This review is very short but it helps us to understand what SLOS looks like and what SLOS is supposed to do. Nonetheless, this book is not about *what* OS is but mostly about *how* to implement embedded OS concepts. If you already familiar with the operating system concepts, you can skip the chapter 1.2. For the detailed description on modern operating systems, there are many good books and you can refer [1] and [2] or you can find many good articles by googling.

1.2 What does Operating System do?

As depicted in Figure 1-1, most of the operating system jobs are handling interfaces between hardware and software. Software applications can be any programs developed by any developers who don't have any idea about the system that runs their applications. How can the operating system interconnect its hardware resources to the unknown software applications? To achieve this goal, operating system virtualizes its hardware. While operating system exposes the system resources to the upper layers and connects them with software applications, operating system abstracts the system hardware such as CPU processor, memory, persistent disk storage so that user application can access without knowledge on the hardware details. There could be other ways, but most operating system applies virtualization to its management area listed in below.

1) *Process Management*
2) *Memory Management*
3) *Storage Management*
4) *Peripheral Device Management*

As the operating system gets matured, the virtualization level also increases. High level

OS such as Linux supports abstracted device driver framework, libraries, SDK for application's usage and so on.

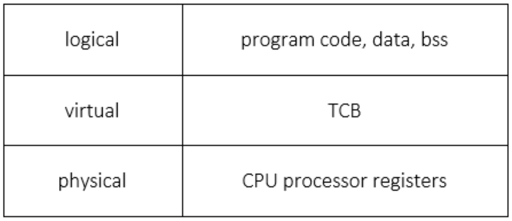
Besides these core areas in upper list, operating system does more. Most of high-level OS are working with networking, power management, security and so on.

1.2.1 Process Management - CPU Virtualization

OS abstracts the CPU processor to run unknown applications. CPU virtualization allows software applications to use the CPU processor without knowledge on the processor (compiler also has its own role on the CPU virtualization). Although there are hierarchical layers in figure 1-1 and they all work together, only one program code runs in a core processor at any moment. In other words, CPU core processor is shared through all programs running in the system including the operating system itself. When you double click an executable program to execute in your computer, operating system allocates a processor to that program, load it into memory and switches to the program. Since multiple programs want to use the CPU processor, operating system should do well in arbitrating all requests from user programs and operating system itself. For this purpose, *scheduler* of operating system determines which software will occupy which CPU processor at some specific moment and how long that occupation will last.

A good application has the ability to split its work into multiple small job instances to use the CPU processors properly. This small job instance is called as *'task'* or *'thread'*. Task or thread is used as same concept in this book and notice that we are going to use these terms interchangeably. The task or thread is an atomic instance running on a processor core, an atomic unit for a scheduler work and also an atomic instance of a process management.

Operating system virtualizes CPU processor by using a task and a task is defined by *TCB (Task Control Block).* The TCB is a big structure containing all information about the task. Sometimes, it has information more than task usage. For example, the *struct task_struct* in Linux has the information not only about CPU virtualization, but it also contains memory allocation list, open file list, locks and so on. It is defined in the *include/linux/sched.h* file of the Linux kernel source tree. If you have a Linux source, open that file then you can find long lines of the definition of Linux TCB (struct task_struct). Anyway, it has all information to run a task on the CPU processor. Nonetheless, the core things needed for the TCB to virtualize the CPU processor are the member variables containing the snapshot of the current CPU's execution context. The snapshot of CPU execution context means normally the registers of the CPU processor.



logical	program code, data, bss
virtual	TCB
physical	CPU processor registers

Figure 1-4: Layer model for processor virtualization

Figure 1-4 describes the hierarchical layers of CPU virtualization. In the layer model, physical layer is for physical hardware electronics. In figure 1-4, the layer 1, physical layer, must be the CPU processor hardware and its registers. These registers are the snapshot of the processor's runtime context.

Virtual layer is the abstraction of the hardware accomplished by operating system. The TCB in the kernel could be used for this layer. The logical layer is for the logical application implementation by using the virtualization of the hardware. The logical layer can be the program code, executables running on top of hardware virtualization. We will also do the same analysis on memory management and storage management.

1.2.2 Process Management - Concurrency

Operating system has a scheduler that arbitrates the requests from each task for CPU time allocation. When there are multiple tasks waiting to be run, the scheduler allocates the CPU processor to the task based on its scheduling policy. Let's run a simple example application to demonstrate what the scheduler does. Below is an example application which prints the application name and then sleeps for one second. Save this source as t1.cpp.

```
#include <iostream>
#include <unistd.h>
using namespace std;
int main(int argc, char **argv)
{
        int i;
        for (i = 0; i < 10; i++) {
                cout << "I am t1 " << endl;
                sleep(1);
        }

        return 0;
}
```

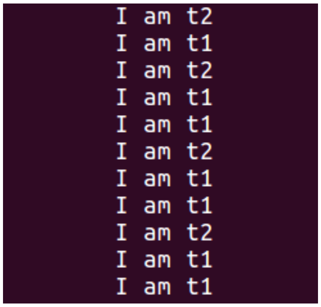
Copy this file and rename it as t2.cpp. Change the t2.cpp to have two seconds sleep and change it to print name “t2” and to run for five times. Compile them with below commands.

shell $ g++ -o t1 t1.cpp
shell $ g++ -o t2 t2.cpp

Then, run the executables all as below.

shell $ ./t1 & ./t2 &

The result of running is shown in figure 1-5. As you can see, application t1 runs at 1Hz and application t2 runs at 0.5Hz for 10 seconds. In other words, application t1 gets the CPU processor every one second, application t2 gets the CPU processor every two seconds. As my test computer has multi core processors, there is a chance that those two applications run in a different processor and there is no issue to execute these two programs at the same time. But even the same test with a single core computer should display the same result. You can also create these tasks more than the processor core number like up to t10 and run them all at the same. Even in this test, you still see the same result. How these two applications can be run in one processor core simultaneously? Scheduler does this magic.



```
I am t2
I am t1
I am t2
I am t1
I am t1
I am t2
I am t1
I am t1
I am t2
I am t1
I am t1
```

Figure 1-5: Running application t1 and t2

Scheduler has a scheme to maintain the TCBs for task t1 and t2. While t2 is in sleep, and t1 sleep time is over, then scheduler wakes up the t1 and activate the TCB of t1 into the processor and vice versa. Task t1 and t2 can share one processor by scheduling in this way.

Another important concurrency issue rises while multiple tasks try to access a common resource. That common resouce shared by multiple applications could be a hardware resource that operating system is currently managing. For example, imagine a situation that task t1 and task t2 are trying to print a message into its stdout, common uart serial port. While task t1 is printing its message, the other task t2 can interrupt the t1, and the message could be mixed between t1 and t2. Then, the output message could look like below.

I am t1

```
I am I am t1 t2
...
```

This example seems not such a critical use-case for the operating system running, but in most cases, sharinga common resource for multitask is very critical and a concurrent access to that shared resource could result in a serious problem in operating system running. Even fixing this type of issue is sometimes very hard.

Another common concurrency issue happens in multitask programming like a global variable shared with multiple tasks. But that is about an application programming technique and is not our concern.

Operating system should properly handle the concurrency problem occuring on shared hardware resources and sometimes support its way to handle common resource to user application's usage. There are diverse concurrency methods such as *semaphore, mutex, spinlock, critical section* used in modern operating systems. For detailed description about concurrency, you can refer [2].

1.2.3 Process Management - Privilege

Another thing we should catch in the simple example of chapter 1.2.2 is that these tasks need to use system hardware resources such as printing message into the screen and a timer to sleep. As mentioned earlier, operating system exposes the system hardware resources to the user applications. Application t1 accesses the screen display hardware and a timer clock hardware for its proper operations. 'cout' is a C++ *STL (Standard Template Library)* library and 'sleep' is a *system call* to access timer clock hardware. Any kinds of user applications running on top of operating system are not allowed to directly access to the system hardware. They always ask operating system to serve system hardware for them. This is an important and basic design policy in operating system. Imagine what happens if unsecure user application access system hardware without any limitations. It can crash the system, it can steal important private information, and ruins other application's data. So, all accesses to the system hardware resource from application are processed by operating system through system call. The system call is the channel for user application to access the system hardware.

Operating system is also one of the softwares in the system and only operating system handles the hardwares. Then, how can the hardware know the different accesses; one is coming from operating system and another is coming from user developed application. Most CPU processors support levels of privilege which can limits the access to the hardware. For demonstration, ARM processor has privilege levels and operating system always runs in *privileged mode* of ARM processor. User applications run only in *non-privileged mode* of processor. So, the processor itself doesn't allow the user application to access a hardware directly. System call is the channel for this.

When user application runs a system call to use hardware resource (e.g. *sleep* function in the t1 task), and when this system call reaches to the operating system, this function makes the ARM processor change its mode from non-privileged mode *(user space)* to privileged mode *(kernel space).* After changing the mode of a processor, the operating system manages the timer clock hardware. After finishing the sleep system call in operating system, operating system goes back to user application and the processor mode is changed back to non-privileged mode. With these steps, the operating system keeps user applications from directly accessing to the hardware and protect system resources from unknown applications' access.

1.2.4 Memory Management - Virtual Address

Normally, it is easy for a program (either user application or operating system) to access memory; it can read/write data from memory just by sending the address to the memory controller. Running a program needs a memory for allocating the program's code, data, stack, and heap. A code memory stores the compiled program instructions. The program counter goes through the code memory while the program is running. Data memory is also a stroage for the data generated in compile time. The memory location to store code and data is determined in the link time of the program build. Stack is a memory storage for the processor's usage such as storing current local data before jumping to another function or storing local array allocation. So, stack is automatically managed by the processor. Application doesn't recognize it. But heap is a memory storage for the application program's needs such as storing image data. Application has the role of allocate and free of it.

Some simple operating system doesn't have a complex memory management. In that case, operating system just services heap memory for the application's requests. A simple memory space for one program running in one processor without memory virtualization looks like figure 1-6.

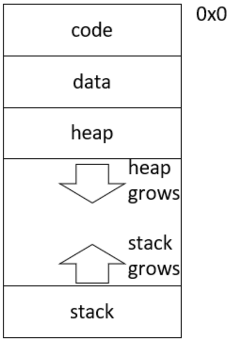


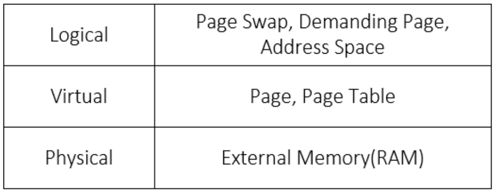
Figure 1-6: A simple memory space example

Operating system manages only the heap memory in this simple case. The heap and stack size are growing depending on the program's runtime status, and they would break into each other's memory region if we don't design the memory space properly. For example, we design the 1KB stack size for a task A and task A declares an array bigger than 1KB, it means a task A's stack overflows, which could result in breaking the whole program running.

High level operating system like Linux does virtualization of the memory. It can abstract the memory space with the help of *MMU (Memory Management Unit).* MMU is placed between CPU processor and external memory. After MMU is enabled, whenever a program running on the processor tries to access to external memory, MMU converts that program's address into other address space. We call the address generated by the application as *virtual address* and the address generated by MMU as *physical address*. MMU abstracts the memory by using a *page table*. Page table is a look up table that has mapping entries from virtual address to physical address.

Figure 1-7 is the layered description of a virtualization for the memory management. Memory hardware device is the physical layer of a memory management. Operating system splits the physical memory by contiguous blocks for its convenience. This block is named *memory frame*. After MMU is enabled, MMU sees the physical memory through page table. The page table and page are the virtual layer of memory management. On top of virtualization through the MMU, operating system can support fancy logical features like below.

1) Virtual address space
2) Paging on demand
3) Process swap to disk



Logical	Page Swap, Demanding Page, Address Space
Virtual	Page, Page Table
Physical	External Memory(RAM)

Figure 1-7: Layer model for memory virtualization

As references in [1] and [3], there are different kinds of address space used in operating system.

1) *Logical Address Space*

   This address space is generated by the CPU. So, when CPU executes a program, all address generated are logical addresses.

2) *Virtual Address Space*

   When MMU is enabled, memory is virtualized through MMU. Then, the logical address used in the processor turns into virtual address; virtual address is same as logical address in this case. MMU needs a *page table walk* for address translation before accessing a physical area of memory.

3) *Physical Address Space*
   This is the address seen by memory unit. If MMU is not enabled, physical address space is same as logical address space.

4) *Bus Address Space*
   This address space is used by peripheral devices whose registers are accessed by *memory mapped I/O*.

When virtual address space is used, each application can have its own address space. Figure 1-8 is an example of separate virtual address space among multiple applications with a shared monolithic kernel. In Figure 1-8, kernel has only one instance of virtual address space with 1GB size from 0xC000_0000 to 0xFFFF_FFFF. But each application has its own virtual address space of 3GB size from 0x0000_0000 to 0xBFFF_FFFF. Application's virtual address space is also called a *User Space* and kernel address space is called a *Kernel Space*. Figure 1-8 shows there is only one kernel virtual address space. All applications share one instance of monolithic kernel. In addition, every application doesn't know there are other applications running in the system. Each application recognizes it uses the whole 4GB address space combined with the kernel. By separating the applications' virtual address spaces, one application can't invade the other application's memory space. This is essential for system protection and stability. Application A's malfunction doesn't prevent application B from running properly or any application can't access other application's important memory data.

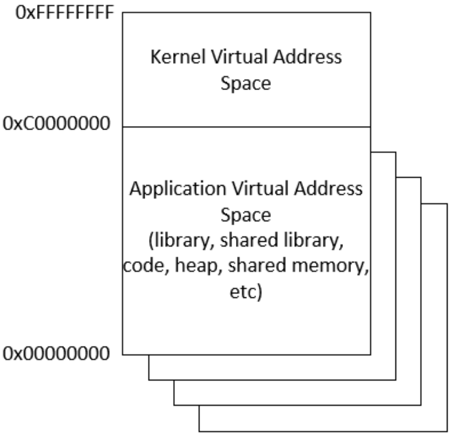

0xFFFFFFFF
Kernel Virtual Address Space
0xC0000000
Application Virtual Address Space (library, shared library, code, heap, shared memory, etc)
0x00000000

Figure 1-8: A virtual address space example for monolithic kernel and applications

1.2.5 Memory Management – Page Swap

Abstracting the memory hardware makes it possible to swap the memory of an application to the storage device. If system memory space isn't enough, operating system could experience a short of memory at some moment. In this case, operating system can temporarily swap some process's memory space to the storage device which is normally much bigger than the memory capacity. When there is not enough memory to load another process, operating system determines which process is going to be swapped out to the storage device. It could choose the candidate application simply based on the *LRU (Least Recently Used).* If operating system needs to rerun the swapped process, the application in the storage is swapped back into the memory again and is restarted. Normally the HDD (Hard Disk Drive) is used for this persistent storage. Now that this HDD's I/O speed is much slower than RAM memory speed, swap-out/swap-in makes some impact on the runtime performance. But there is no other way when RAM space is used up. In addition, recent persistent storage like NVMe SSD is very fast; up to a few GB/s speed.

The process swap looks like figure 1-9.

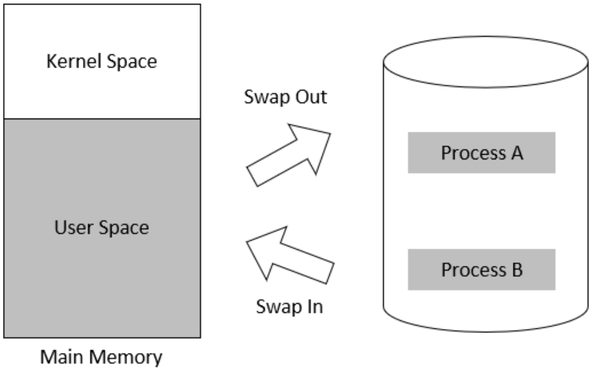


Figure 1-9: Swapping two processes with storage device

Since the storage device is very slow compared to memory speed, rerunning the swapped process takes much longer than running the process from the memory. If operating system enables the demanding page feature, only part of the process can be swapped out and in. When application needs to be started, only core part of the application can be loaded into memory. This makes it faster to launch a large application. Even though it is slow to swap in, there are many advantages and the main merit of swapping is that it enables the operating system to have more memory than it has in the system.

1.2.6 Memory Management - Page

After MMU is turned on, operating system starts to manage the RAM memory with *page*.

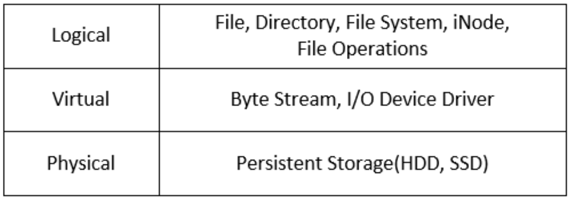
Page is a contiguous memory block which is accessed as a single entry of a page table. The page size is architecture dependent. For example, ARMv7 architecture describes page size as *small page* (4KB*), large page* (64KB), *section* (1MB) and *super section* (16MB). Since page is a minimum access unit of an operating system, the page size is also the mimimum memory allocation size by the operating system. In other words, if an application allocates one byte of memory, operating system allocates one page which is much larger than the allocation size needed. In this case, the rest part of the allocated page is wasted. This is called as an *internal fragmentation* and a bigger size of page has a worse internal fragmentation. Instead, large size of page has lesser page fault and more *TLB (Translation Lookaside Buffer)* cache hit than small page size. This makes large page size can be faster than small page size in memory access. TLB is a cache of a page table for the enhancement of the translation speed of virtual address. Since every virtual address needs to be translated through the *page table walk* and most access to the main memory is pretty localized, caching the page table entry is good for performance. In words, small size page has lesser internal fragmentation but worse performane than larger page size.

Operating system partitions the physical memory with *memory frames*. Page table and page are the virtual part of the physical memory frames. Memory frame is directly mapped to memory. For demonstration, if frame size is 4KB, the first frame covers 0x0000_0000 ~ 0x0000_0FFF of physical memory, the second frame covers 0x0000_1000 ~ 0x0000_1FFF and so on. But the page can be mapped to any place (memory frame) in memory through the page table. Even it is possible that the same virtual address from different application can be mapped to different physical address. Same virtual address doesn't need to be the same physical location of memory and all this tricky mapping is done through MMU and page table walk.

When there is a memory allocation request from an application, operating system can put off the real allocation of physical memory until there is a real physical access to that memory. This is called *Demand Paging* or *Lazy Allocation*. For example, an application A requests a heap of one-page size, operating system just records it in its virtual memory pool but still it doesn't allocate the 4KB physical memory yet. When application A tries to access that heap memory while it is running, for example write an integer value to the heap, then operating system allocates the physcial heap memory for the access of the application. In addition, like chapter 1.2.5, operating system can allocate more memory by swapping in-and-out the pages to the disk storage.

1.2.7 Persistent Storage Virtualization

Persistent storage device is also an important hardware resource that operating system should manage. There are many different types of persistent storage devices; Hard Disk Drive (HDD), Solid State Drive (SSD), Embedded Multi-Media Controller (eMMC) and so on. Even the technology keeps evolving (as of now 2021, really fast persistent storages are launched in the market; NVMe and intel optane storage. These are still SSD storage.). Figure 1-10 depicts the layer model for persistent storage management. Persistent storage's virtualization needs a device driver to interface with diverse disk media. These different drivers perform I/O operations with the corresponding persistent storage. In RAM memory management, operating system doesn't have a specific device driver for memory access. Operating system can directly interact with memory controller with memory address. But, since disk storage is very slow, and there are different types of storage, a specific disk driver needs to be developed to handle the disk I/O transaction and sometimes it needs to handle the interrupt from the disk. On top of this I/O device driver that virtualizes the physical persistent storage, the operating system can perform its logical implementations of the persistent storage such as *file system, file,* and *directory*. File systems are also developed in many different ways. Examples of file systems are *NTFS, VFAT, EXT2, EXT3, EXT4* and there are more.



Logical	File, Directory, File System, iNode, File Operations
Virtual	Byte Stream, I/O Device Driver
Physical	Persistent Storage(HDD, SSD)

Figure 1-10: Layer model for persistent storage virtualization

File operations such as file read, write, delete, rename provide a common, abstracted way for the applications to use the physical storage disk. So, there is another virtualization layer on top of this logical implementation layer. This layer is called a *VFS (Virtual File System)*. We will recall this in chapter 6, storage management. Even the storage devices installed in the system are different, applications can access to the storage in a common way of virtualized file system operations. This is the reason why we can run same file operations (file read, write, create, delete) in the different computers which has different types of disk storage.

1.3 Then, What SLOS does?

High level operating system does more than the management described in chapter 1.2. It has a well-defined device driver framework for peripheral hardware devices and it also supports security, network and so on. If there is a new hardware in the system, operating system should manage it and mostly, it needs to connect the functional features of that hardware to the user applications.

In this chapter, we are going to have a short skim on what SLOS does and compare it with the commercial operating system. As shortly mentioned earlier, SLOS isn't a commercial maturity operating system and is developed for fun and for studying the basic concepts of operating system.

Compared to the commercial operating system features, SLOS has below features.

1) Simple Process Management
2) Simple Memory Management
3) Simple Ramdisk Storage Management
4) A Device Driver for Custom-designed Hardware

These features in SLOS are simple and have some limitations. For example, maximum file size supported in *SLFS (Simple and Light File System)* is about 63KB (But we can increase this size anyway). SLOS also is not tested very well. I myself developed those features from scratch and tested their operations. But this is alright for hobby project and for studying purpose (Even my XBox game crashes very often!).

1.3.1 SLOS Process Management

As a process management is a management for a CPU processor, SLOS process management is implemented on how to allocate ARM processor to the applications. The target board we are going to work with has *ARM Cortex-A9* dual core processor. SLOS has a simple TCB structure for virtualizing this ARM processor. SLOS is designed and tested for ARMv7 architecture, and the TCB of SLOS abstracts (virtualizes) the run time snap shot of the context of ARM processor. SLOS process management has a *forkyi()* function to fork another task and a new TCB for the new task is created in the forkyi() function. These SLOS tasks have two states; *TASK_RUNNING* and *TASK_WAITING*.

For managing tasks, SLOS has a *CFS (Complete Fair Scheduler)* scheduler and a *soft realtime scheduler.* The implementation of CFS scheduler immitates the implementation of Linux CFS scheduler. It has a *sched_entity, virtual runtime* concepts like Linux does. It also uses the *red-black tree* structure for its optimized handling of timer framework. The implementation file of red-black tree is directly copied from Linux source file. Handling red-black tree uses the *run_node* like Linux. The idea of CFS scheduler and its implementation in SLOS are borrowed from Linux. When I first start this project, the Linux version is 3.10, but Linux changes very fast. I am not quite sure current Linux still has the same scheme.

A soft realtime scheduler is also implemented in SLOS. It is an *EDF (Earliest Deadline First)* real time scheduler which picks the most urgent task for the next run task.

The heart of these scheduler implementation is a *timer framework*. SLOS timer framework maintains a timer tree which is updated at every timer tick interrupt. There are three types of timer tick in SLOS; *sched_tick* for CFS scheduler and *realtime_tick* for EDF scheduler, and oneshot tick for oneshot task. Actually, the sched_tick itself is one of realtime_tick. The CFS scheduler and EDF scheduler normally use a periodic timer tick but oneshot tick use only one timer tick. oneshot_tick is used for sleep functions later.

Besides the CPU processor, the process management of SLOS needs below hardwares for its proper operations.

1) *Timer hardware*
2) *GIC (Generic Interrupt Controller)*

Timer hardware is necessary to give a time information for managing the timer framework. It provides time information to update the timer tree of each task and is reprogrammed for the next timer interrupt. GIC is an ARM interrupt controller and receives interrupt signals and distributes it to the processor. These two hardware modules provide a heart beat to the operating system so that the operating system can keep updating the timer tick information in the timer framework and can schedule next job in time. The driver implementations of these two hardware will be covered in the chapter 4.

SLOS can also load user applications. SLOS has a simple *ELF loader* to parse, load user application to a specific memory location and jump into the entry point of user application. After running, the user application can access the system hardware resource by using a SLOS *system call*. Currently, SLOS syscall supports only *write()* function which writes character text from user memory buffer into UART serial terminal. By using this system call, user application can access UART hardware and print message into the screen terminal. This user application and elf loader is covered in chapter 6 storage management with a ramdisk storage.

Since multiple tasks can run at a time, there is a *spinlock* implementation in SLOS for task synchronization. A shared resource can be synchronized among multiple tasks' accesses by using the spinlock. For example, manipulating the runqueue is protected by spinlock because another task can be enqueued after a timer interrupt while current task's enqueuing isn't completed. Spinlock also can be used to synchronize the accesses from multiple CPUs.

SLOS is evolved to support the *SMP (Symmetric Multiple Processor)*. SLOS also supports *SGI (Software Generated Interrupt)* to implement the message queue for *IPC (Inter Processor Communications)*. SMP and SGI implementation is covered in the chapter 8 separately.

As SLOS (Simple and Light OS) name represents, SLOS process management is simple and limited in its features; SLOS doesn't support non-privileged user mode execution. In other words, all tasks including user applications run at privileged mode, the kernel mode. All tasks including user applications run in *supervisor mode (SVC mode)* of ARM processor and can access all hardware resources. SLOS doesn't support the switch between user mode and kernel mode. But this is not an issue in our case because SLOS is not a commercial OS and a hobby project and developed out of personal curiosity.

In summary, SLOS process management supports below features.

1) Simple TCB for ARM processor
2) CFS scheduler

3) Soft realtime scheduler
4) Timer interrupt and timer framework for scheduler tick
5) Fork to another task
6) System call
7) Concurrency handling with spinlock
8) Dual core SMP
9) Task state; TASK_RUNNING, TASK_WAITING

There are still many features which are not yet supported by SLOS. As mentioned, switching between usermode and kernel mode, semaphore and many more features.

1.3.2 SLOS Memory Management

SLOS has a simple memory management; it enables memory management unit (MMU), creates page table, does virtual address translation and lazy memory allocation by handling a page fault.

SLOS creates 2 level page tables for small pages. Small page size is 4KB. There is a secondary bootloader *(SSBL)* in SLOS that creates page tables before bootup and enables the MMU. After enabling MMU, SLOS address in ARM processor (logical address) turns into a virtual address that needs a translation before accessing physical memory. So, SLOS supports virtual address by enabling MMU and page translation table.

SLSO has 64 MB heap memory. Heap memory manager does a *lazy allocation* which is applied only to allocates heap memory when page fault happens. Page fault happens when a virtual address which doesn't have a valid entry in the page table is referenced.

SLOS doesn't support user application's virtual address space. Every task including user application shares one translation table.

SLOS memory manager supports below features.

1) 4GB virtual address space
2) 4KB small page table
3) MMU and page translation table walk
4) Page fault handler and demanding page

SLOS memory manager has the following limitaions.

1) SLOS memory manager doesn't support different *ASID (Address Space IDentifier).* This means all tasks including user task shares same virtual memory space.
2) Only heap memory has different virtual address and physical address. Other memory region has a *direct address mapping* which has the virtual address same as physical address.

1.3.3 SLOS Storage Management

SLOS doesn't have a I/O device driver for persistent storage. Instead, a simple storage management is demonstrated with the ramdisk storage. Ramdisk is not a persistent storage, its contents go away if power is turned off. But by using ramdisk, SLOS can easily immitate a storage device on top of memory read/write operations. A good thing of this is that we don't need a complex disk I/O driver for developing SLOS's virtual layer of storage management as in figure 1-10.

On top of ramdisk I/O, SLOS implements *SLFS (Simple and Light File System).* SLFS has features that creates a file, reads/writes data to a file, mounts file system and formats file system. SLFS has a meta data block located at the first part of the disk. SLFS uses *inode* to describe a file. Inode associated with a file is a part of the meta data block of the storage media.

Current SLFS doesn't support directory and blocking I/O. Supporting blocking I/O is a feature of I/O device driver. Since the disk storage is way slower than the RAM memory, the blocking disk I/O driver blocks the caller task into *waiting state* until the I/O request is completed. When I/O request is finished in the disk, it generates an interrupt to wake up the waiting task and have the task resume its job. SLFS doesn't care the blocking I/O since it only has ramdisk memory I/O.

SLOS can load a user application from the ramdisk. SLOS has an ELF parser. This is not a part of storage management, but loading and running user application from ramdisk is also touched in the storage management. By using SLFS and ELF parser, SLOS reads a user application executable from the file stored in the ramdisk, interpret the ELF header of the application, and loads it into the memory properly, then runs it.

In summary, SLFS supports below.

1) File Creation, File Read and File Write
2) ELF loading and fork user applicatioin

SLFS has limitations as below.

1) It doesn't support directory
2) A file size is limited to about 8MB
3) It doesn't have blocking I/O of the storage

1.3.4 SLOS Device Driver

SLOS doesn't have a well-define device driver framework. But it touches basic concepts on developing a device driver for custom peripheral hardware devices. Operating system can talk with its peripheral device with interrupt and registers. Normally, interrupt is coming from hardware to tell the software that there is a hardware event for software to handle. Register is used for both (hardware and software) directions. Software can tell the hardware by using control register and config register, and the hardware updates its status register to tell the software on its current status. There are a bunch of registers in the hardware to store the hardware information or to set the hardware control configurations.

SLOS uses three peripheral hardware devices. It uses *UART* to print message into serial terminal or get user input from the terminal. Another peripheral hardware is a custom hardware *(Modcore coprocessor)* which is developed in programmable logic subsystem to perform a simple math operation on the memory data. It also uses a custom *DMA (Direct Memory Access)*. The third hardware is an outstream device also running in the programmable logic subsystem. These custom hardware devices are developed with *VHDL (Very High Speed Integrated Circuit Hardware Description Language)* in a programmable logic of Zynq7000 chipset. The device driver for UART is already included in Xilinx library and directly copied into SLOS source tree without modification. We will not develop UART driver (it's not our interest) and just port the Xilinx UART driver to SLOS. SLOS prints message and gets user input through this UART device and its driver. But we will develope our simple Modcore coprocessor and Outstream device by defining the registers, interrupt and develop their devcie drivers in SLOS.

1.4 Top View of SLOS

As described, SLOS has following features.

1) *Process Management*
   a) Timer framework and timer interrupt handler
   b) Task control block and fork
   c) CFS scheduler and soft realtime scheduler
   d) System call
   e) ELF Loader to load User Application
   f) Task synchronization through spinlock
   g) SMP
   h) Task state: Running and Waiting
2) *Memory Management*
   a) Small page table
   b) Enabling MMU and page translation table Walk
   c) Virtual memory space and demanding page
3) *Storage Management*
   a) iNode and file operations
   b) SLFS file system
   c) Ramdisk containing user-built application
4) *Device Driver*
   a) Custom peripheral hardware in programmable logic

DMA device driver, Outstream device driver and their interrupt handlerThe exception handler is the center of all these operations. It handles interrupts, system call, page fault in addition to the reset handler. All these SLOS's management modules work on top of the exception handlers. Figure 1-11 shows the top view of the whole SLOS submodules. As you can see, the exception handler is placed in the center of all these blocks. Blocks related to process management are irq handler, timer framework, scheduler, tasks, ELF loader, context switch. Blocks regarding memory management are page fault handler, virtual memory manager, MMU table walk, kmalloc/kfree. Storage management blocks are ramdisk, SLFS file system. Device driver blocks are DMA device driver, Outstream device driver, and UART device driver. There are also hardware blocks such as a GIC, timer, Modcore coprocessor, Outstream device and UART. The gray-ed blocks are running in the CPU 1 for SMP SLOS. The implementation of these blocks is covered from chapter 4. We will add blocks one by one from chapter 4.

SLOS is being developed and being tested on Xilinx ZC702 Evalution Board. This evaluation board has a Zynq7000 chipset which has Cortex-A9 dual core processor which is based on ARMv7 architecture for its *Processing Subystem (PS subsystem)*. The PS subsystem has also periperhal devices such as UART, SDIO, I2C, Ethernet and many GPIOs. Most of them are not necessary for SLOS except UART. In addition, Zynq7000 chipset has an FPGA *Programmable Logic Subsystem (PL subsystem).* All SLOS modules except VHDL modules for Modcore coprocessor and Outstream device run in the ARM processor of Processing subsystem. The detailed VHDL implementation for Modcore coprocessor and Outstream device are not covered in this book, but adding custom IP to Zynq7000 chipset and software-hardware codesign will be explained in the chapter 7. Since there are not Zynq7000 specific code in SLOS, these SLOS submodules except custom hardware part also can be run on other ARM chipsets based on ARMv7. Some part of this book is also developed through custom board with Zynq chipset. In this case, only the UART configuration changes. You can extend the concept in this book to other ARM chipset board. Actually, SLOS was first developed and tested on *Qualcomm Snapdragon msm8x60* chipset and recently ported into Xilinx Zynq 7000 chipset.It had worked with *LK bootloader* (Google Android bootloader) long ago but has not been maintained for a while and now seems to be broken. The SLOS features in msm8x60 is not supported any more.

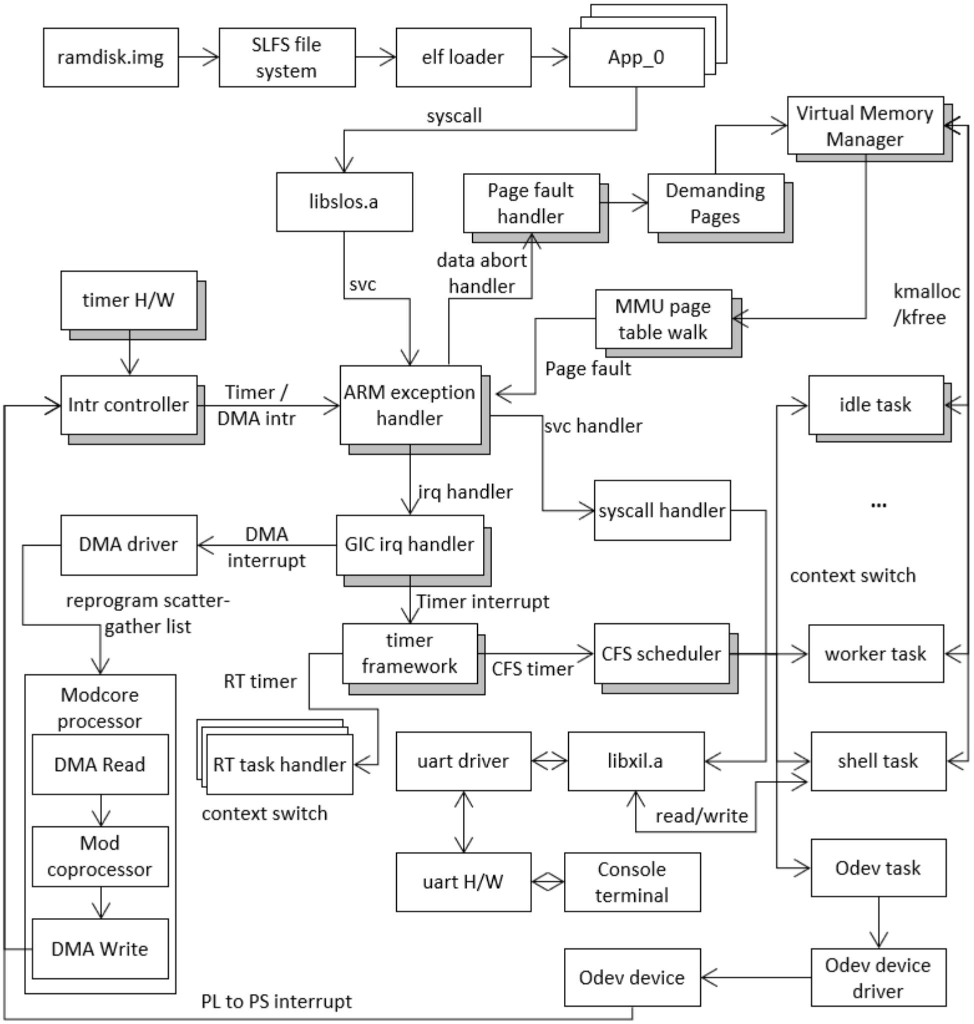


Figure 1-11: A top view of SLOS all subsystems and their control flow

1.5 SLOS Version Control

SLOS source files are managed by using *Git*. Git is a distributed verson control tool for tracking changes in files and coordinating work on those files among multiple developers. Git was created by *Linus Tovalds* in 2005 for the developement and maintaining of Linux kernel [4]. Git is widely used for version control and even Microsoft started to support Git in its Visual Studio. Git is easy to install and free to use. There is a free Git installation binary running in MS Windows OS, or if you are an Ubuntu user, you can simply run *"sudo apt-get install git"* to install Git. I prefer using a command line interface but there is also GUI based

Git interface in Windows Git. If you are not familar with Git tool, there are many good articles on the web or refer to a book in reference [5]. Before going further, let's have a quick review on Git needed to follow the SLOS development process. This review is short but should be enough to work with Git by yourself in this book.

1.5.1 Git as a Distributed Version Control System

Git is a *DVCS (Distributed Version Control System)* that all client users have a fully mirrored copy of the central repository. In this case, central repository is just one another copy of a source version that is shared to all client users. When server repository is broken, it can be easily recovered by one of client's source Git repository. In addition, when server is down, you still can work on your local Git repository and push your changes after the server is back. As an example of figure 1-12, three users have its own local Git repository and share their changes in the central repository. They can push their work through any kind of application layer protocol. Any type of application protocol of communication layer can be used for this purpose. For example, *ssh, git, http* or even MS Windows network directory path can be used for this communication.

In figure 1-12, each user can update his changes to central repository with below two-phase steps.

1) Manage developer's local repository

2) Synchronize developer's local repository with central repository.

The management of local Git is composed of two stages; *staging area (or index area)* and *commit area*. You can keep adding or deleting files to the local staging area whenever you change files, create files or delete files. After all changes are done to the staging area, then

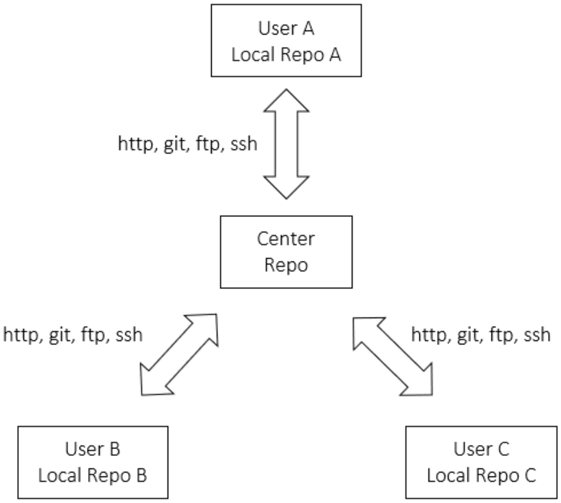


Figure 1-12: Distributed Git Example

you can bundle those changes by a *commit* command. Commit is the final step to store your changes into your local Git repository. Following commands are used for managing local Git repository.

1) *git status*

   This command shows the different status of files between the working tree and staging area. It lists up the modified but not staged or staged but not yet committed. Normally, this is the first command for you to see the status when you start to commit your changes.

2) *git add*

   This command adds the changed file to staging area. You can still edit this file before committing it. If you delete files by shell command (/bin/rm), then add -u option to stage the changes of file deletion.

3) *git commit*

   This command commits the changes in staging area to local Git repository.

4) *git log*

   This command shows the recorded history of commits. There are some options to customize the showing information.

5) *git diff*

   This command shows the changes of files. If it is used without any additional parameters, you can check the current changes before staging those files. This command also can be used to see the whole changes between two commits.

6) *git branch branch_name*

   This command creates a new branch which will make a different history tree of commits. Branch is a separate commit tree that has a distinct change history. For example, branch A could have versions for project A and branch B has versions for project B.

7) *git checkout branch_name*

   This command moves to different branch from current working branch. This command switches back and forth between branches.

8) *git reset*

   This command moves the *HEAD* pointer to previous snapshot. With this command, you can reset your wrong changes or delete unwanted commits. Be careful to use this command with option --hard which remove the history permanently. If you want to remove all your local changes, *"git reset HEAD --hard"* is useful command to do that.

9) *git clone*

   This command is used to clone the remote repository for the first time. It pulls from remote repository and check out master branch.

10) *git push*

    This command pushes local commits to remote repository.

11) *git pull*

    This command fetches changes from remote repository which is made by coworkers and merges those changes to local repository.

These commands are necessary while you are following the chpaters of this book. Commands from 1) to 8) are necessary for managing local Git which is corresponding to phase 1. Commands 9) to 11) are for handling the interaction between local Git and remote central Git, which is about phase 2. Figure 1-13 illustrates how those commands work in each phase. The left 3 blocks in figure 1-13 correspond to the phase 1 handling the local repository. Handling between the right-most block (Remote Repo) and other 3 blocks in figure 1-13 is phase 2 Git commands.

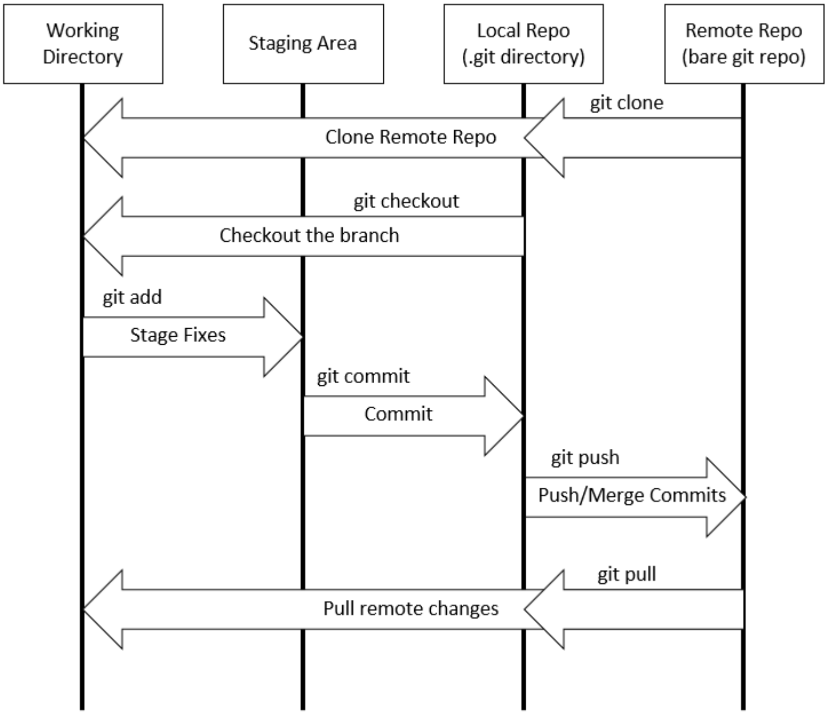


Figure 1-13: How Git commands work

There are many good Git commands to remember such as *git show, git stash, git tag, git merge, git rebase, git remote, git fetch, git config*. Each of these commands also has many options for different operations. For more of these, refer [5] or you can find many helpful articles by googling. For working with SLOS, remembering the upper 12 commands is enough.

1.5.2 SLOS Git Repository

All source code and change history of SLOS are uploaded into the public repository. Anyone can access it. Check below link for SLOS source code.

https://github.com/chungae9ri/slos

There are many branches in this SLOS central Git. It has branches for each chapter of this book, *msm8x60* branch, *zynq* branch, zynq_custom branch and *master* branch. Except the master branch, msm8x60, zynq, zynq_custom branches are not maintained any more and are deprecated. The master branch has the latest source files for SLOS running on an Zynq7000 chipset. As third edition is written, there is a tag 'v2.0' which has all commits working with chapter 1 to chapter 8. You can download it, try it, and also contribute it by fixing a bug or implementing a new feature. This repository will be used from chapter 3.

Another repository needed to be mentioned is a *tools* repository and *meta-slos* repository. Below link has prebuilt binary images and tools needed for the convenience of SLOS devlepement.

https://github.com/chungae9ri/tools

This repository has following binaries.

1) *BOOT.BIN*

   BOOT.BIN is a boot file for Zynq7000 chipset. It has three partitions. First partition has a Zynq *FSBL (First Stage BootLoader),* second partition has a FPGA bit stream for programmable logic. The last partition is configurable. The last partition could be used either for *uboot* for loading Linux or for a standalone application or other operating system like *FreeRTOS.* BOOT.BIN in tools repository has the simplest "hello world" version of SLOS. This BOOT.BIN can be used to see the SLOS running in the board.

2) *design_1_wrapper.bit*

   This binary is a bit stream for programmable logic subsystem. It is a default, bare bit stream and doesn't have a meaningful functionality for now. This file is necessary to build the BOOT.BIN. If you are interested in operating system only, you don't need to touch this. But programmable logic is very powerful by allowing you to design your custom hardware. We will design our own peripheral hardware in the PL subsystem and develope a device driver running in SLOS in chapter 7.

3) *design_1_wrapper.hdf*

   This is a hardware definition file created by Xilinx. This file is used to create a *petalinux* project. Petalinux is a Xilinx's Linux distribution running on an ARM processor in Zynq CPU. This file is needed to build petalinux project. Petalinux project isn't needed for developing SLOS but needed to build BOOT.BIN.

4) *gcc-arm-none-eabi-6-2017-q1-update-linux.tar.bz2*

   This file is a *GCC compiler tool chain*. This file is running on Linux host and 2017 q1 version is used in this book. You can also download the latest version running for Windows, Linux, Mac from below link. SLOS is developed and verified with 2017q1 version of GCC tools.

   https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads

5) *image.ub*

   This binary is a petalinux booting image for Zynq chipset. We don't use this image but you can experience Linux features with this binary.

6) *kernel.elf*
   This is a "hello world" SLOS booting image. SLOS build system generates this file. This file is placed in the third partition of BOOT.BIN.

7) *u-boot.elf*
   This is a Linux bootloader. If petalinux image (image.ub) is used for operating system, this file must be used for the third partition of BOOT.BIN.

8) *zynq_fsbl.elf*
   This file is a first stage bootloader of Zynq7000. It does basic hardware initialization and download the FPGA bit stream to PL subsystem and jumps to the third partition of BOOT.BIN.The third partition could be either standalone application or u-boot.elf for the Linux image (image.ub) or FreeRTOS or SLOS (kernel.elf).

9) *zynq_fsbl_hook.elf*
   This file is a modified version of FSBL. It has a hook function to load a ramdisk image from SD card to memory. This FSBL is used to develop an SLFS which is covered in chpater 6 Storage Management.

These binary images and GCC tools are for your reference. You don't need to use these images and we will develop kernel.elf and zynq_fsbl_hook.elf while going through this book.

Another good reference repository is meta-slos. It stores *Bitbake recipes* to build SLOS. Following is the link.

https://github.com/chungae9ri/meta-slos

This repository has a full-automated build recipes for SLOS. It first downloads cross compiler tool chains, uncompress it, set it up in your home/bin directory and then downloads the latest SLOS sources and builds it. Currently, it downloads the SLOS v2.0 tag and build the kernel.elf. All of these steps can be done with one command. A detailed description and creating the meta-slos is covered in appendix A. If you are interested in *Yocto,* this could be a good reference.

1.6 Summary

In this chapter, we summarize the major features of modern operating system. Three big pieces of modern operating system are process management, memory management and storage management. We make a layered model for better analysis on these areas. The layered model is composed of physical layer, virtual layer and logical layer. Physical layer is mostly the hardware, electronics itself; CPU processor, memory and storage. Virtual layer abstracts the physical resources for better logical implementations. Logical layer implements the hardware features on top of the abstraction of hardware. So, main job of operating system is virtualization of system hardware resources and connect them to the applications.

The second part of this chapter is a brief summary of SLOS features and a comparison between SLOS and modern operating system. SLOS has also process management, memory management *(SLMM),* storage management *(SLFS).* SLOS also has custom peripheral hardware devices by using VHDL in the PL system and implements their device driver through registers and interrupts. In addition, SLOS supports SMP SLOS in the v2.0. Although those implementations are limited and simple, it helps a lot to understand modern operating system concepts and is enough for this personal hobby project.

References

[1] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, *Operating System Concepts*, Ninth Edition, Addison-Wiley (2012)

[2] Remzi Arpaci-Dusseau, Andrea Arpaci-Dusseau, *Operating Systems: Three Easy Pieces,* Version 0.92, Arpaci-Dusseau Books (2018)

[3] *linux/Documentation/DMA-API-HOWTO.txt*

[4] *https://en.wikipedia.org/wiki/Git*

[5] https://git-scm.com/book/en/v2

2 Environment Setup and Processor Review

2.1 Introduction

Setting up the OS development environment needs involvements from a wide range of technologies. For SLOS development, environment setup is not just a simple IDE (Integrated Development Environment) installation such as Eclipse or Visual Studio. Setting-up the development environment includes a Xilinx Vivado IDE setup, build system, cross compiler, linker script, bootloader, debugger and knowledge on hardware and the processor itself. Sometimes having well-defined development environment is too hard for the beginners to start this fatanstic journey. In this SLOS development process, we don't want to debug the hardware. We don't want to develop a bootloader. Those are non-OS stuffs and we are not interested in. But we should prepare them very well in order not to get lost and nip this fun project in the bud. Chaper 2 and 3 covers these prerequisites, and essential preparations before digging into the real stuffs.

Since operating system is a special software which is driving the hardware installed into the system, especially the CPU processor, knowing the processor is essential to the operating system development. The later part of this chapter covers a necessary information on the ARM processor and Xilinx Zynq7000 chipset. But if you are familiar with ARM processor and Xilinux Zynq7000 chipset, you can skip the chapter 2.5 ~ 2.7.

The setup descriptions and preparations through the chpater 2 and 3 are my specific ways used for SLOS development. SLOS development environments are based on Xilinx 2019.2 version. Xilinx ships their development tools (Vivado and SDK) very often, and Xilinx tool version used in this book looks a little bit old. Nonetheless, you can still use the latest Xilinx tools with the similar way used in this book. There also could be other environments setup methods customized with your own hardware.

2.2 Development Environment Overview

Preparation of development board and PC for host machine is the first step to start custom operating system development. The SLOS development environment of host computer and target board used in this book looks like figure 2-1.

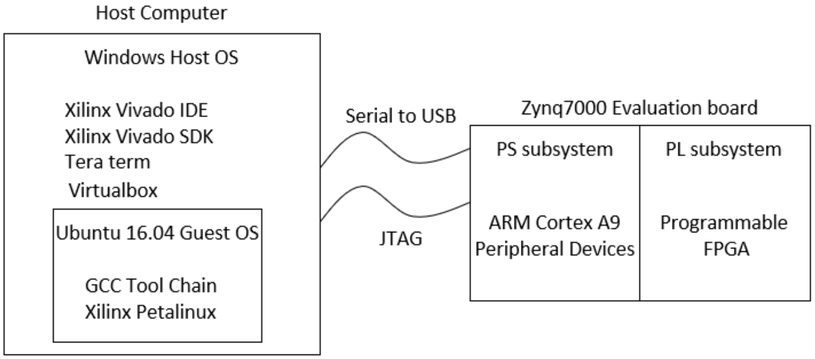


Figure 2-1: Development Environment Setup: host computer and target board connection

Since operating system runs right on top of hardware and drives that hardware to support application software, we first need to define the hardware for the operating system to work on. To demonstrate, process management needs a processor hardware (CPU) which runs different application process, memory management needs an MMU (Memory Management Unit), and file system needs a storage such as hard disk. There are many commercial boards you can use for your custom operating system such as popular *Raspberry PI.* When I first started to develop the custom operating system, I reused my company's development board which has Qualcomm msm8x60 chipset. I also reused the *Android* bootloader and downloader *(fastboot).* By using verified hardware, bootloader, and downloader, I can avoid wasting my time in working on hardware debugging or bootloader development. Those are not operating system stuff but must be solved before starting the operating system development. The board that I used at that time had also well verified JTAG debugger which is very important in this type of software development.

SLOS is not running on emulator or on desktop computer, but it is an embedded OS which runs on the specific target board. So, we first have to choose the proper development board first. This book describes the development process of custom OS based on *Zynq7000 Evaluation Board.* If you are to choose different development board and try to apply the development procedures described in this book, it must have ARMv7 processor, a good debugger, downloader, basic drivers (UART, timer interrupt), and a bootloader which initializes hardware and loads your operating system. SLOS itself can be run in other commercial boards but it is recommand to use the same Zynq7000 evaluation board to run the SLOS properly. Especally, the chapter 7 needs a programmable logic hardware to design custom hardware and its device driver. Nonetheless, the steps and implementations in this book still could be applied to the different board with some modifications.

Development PC must have a cross-compiler to build correct executables for the ARM processor. The cross-compiler used in this book is *GCC (GNU Cross Compiler)* tool chain running on Ubuntu 16.04. Cross compile means creating an executable for a target platform other than the one on which the compiler is running. We run the GCC tools running on Intel processor in host PC and generate SLOS binary running on ARM processor in target board. There are also GCC tool chain running on Windows, but this book assumes GCC tool chain installed in the Ubuntu Linux. If you follow the Appendix A, you can set the toolchain and build at once. But basically, we follow traditional build process step by step and run 'make' command to build the SLOS. In chapter 8, we will use bitbake build.

Ubuntu can be installed to a virtual machine. We are going to use a *Virtualbox* running an Ubuntu 16.04. This book doesn't explain the installation of Virtualbox and the installation of Ubuntu 16.04. Virtualbox becomes very stable and is well supported now. Personally, I don't experience any problems with this setup in developing SLOS so far. You might prefer to using a standalone Ubuntu host computer rather than using a virtual machine. This is also possible for SLOS development and this was my first development environment. But Virtualbox is free, stable and good to interact with host-guest OS, I recommend you to use Virtualbox with Ubuntu 16.04 guest OS installation.

Xilinx development tools such as Vivado IDE, SDK and Petalinux are very heavy tools and consume a large amount of resources in memory, disk and CPU in the host PC. But they have important roles in the custom OS development. Vivado IDE is used to build default bitstream for the PL subsystem and allow us to design a custom hardware. Xilinx Vivado SDK is used for *gdb* server for debugging remote target board, and Petalinux tool is used to create a BOOT.BIN that includes SLOS kernel executable. Installation of these tools are covered in chapter 2.3.5 and 2.3.6. The custom hardware design via Vivado is explained in chapter 7.

2.3 Development PC Setup

2.3.1 Virtualbox and Ubuntu 16.04 Installation

The installation of Ubuntu 16.04 with GCC tool chain and Petalinux is needed for development PC. If you are using Windows PC, you first need to install Virtualbox and install Ubuntu 16.04 as a guest OS into the Virtualbox. This book assumes Windows 10 for host OS and Ubuntu 16.04 for guest OS. The Virtualbox allows the guest OS (Ubuntu 16.04) to connect to remote device through usb, uart to serial connection. It also allows the guest OS to access host OS (Windows 10)'s hardware resource as well. Installation Ubuntu and Virtualbox are not explained in this book. You can refer [4] and [5] for download and installation of them. You can get a lot of informations just by googling as well.

2.3.2 Git Installation

Next, you have to install a version control tool Git into your Ubuntu guest OS. Git is another masterpiece of Linus Torvalds for managing Linux source version control. It isn't only used in many open source projects, but Microsoft Visual Studio also supports Git 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.

A description about Git and major commands needed to follow up this book are already described in chapter 1.5.

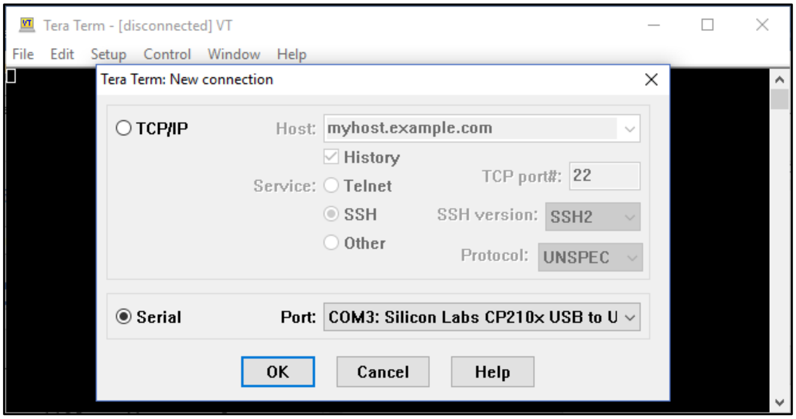
2.3.3 Serial Terminal Installation

After installing Virtualbox, Ubuntu, and Git, we need to install serial terminal. This serial console is going to be used as an I/O interface to the target board. The Zynq7000 board doesn't have any I/O interfaces such as a screen for output message and a keyboard for input. The serial console is going to be used as a user interface instead of them. This console is also used to debug the operating system.

There are many serial communication applications. We are going to use a *Teraterm*. Search a Teraterm on the internet, download it and install it. Steps are quite straightforward.

After installation, run Teraterm program for the first time. Since there is no preloaded connection, Teraterm will ask you a new connection like figure 2-2. If the target Zynq board is on and is connected through usb cable properly, you can see the connection window as figure 2-2. Choose the 'Serial' in the connection window and select a COM port as below figure.

If you don't choose a 'Serial' in the new connection window, then, after starting a Teraterm, go to setup->serial port and set the serial port configuration as below figure 2-3. Port number (COM5 in below figure) is dependent on your system and you have to choose correct COM port number in your system.



Tera Term: New connection

○ TCP/IP

Host: myhost.example.com

☑ History

Service: ○ Telnet

● SSH

○ Other

TCP port#: 22

SSH version: SSH2

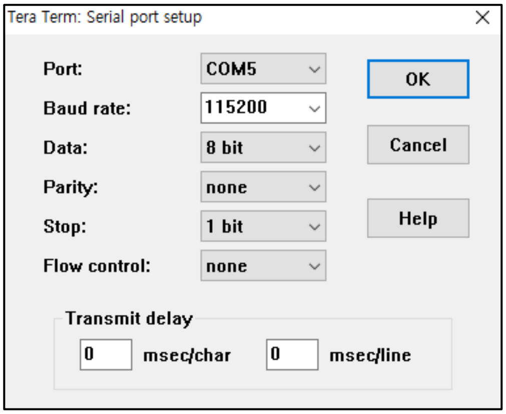
Protocol: UNSPEC

● Serial

Port: COM3: Silicon Labs CP210x USB to U

Figure 2-2: Teraterm new connection window

SLOS always uses a CR (Carriage Return, '\n') for its new line character and doesn't use a LF (Line Feed, '\r'). If you want to see a correct new line in Teraterm console window, go to setup -> Terminal and set the New-line 'Auto' as in figure 2-4. Then, save the Teraterm settings for the next run. Go 'setup -> save setup' and save it with a default name.



Tera Term: Serial port setup

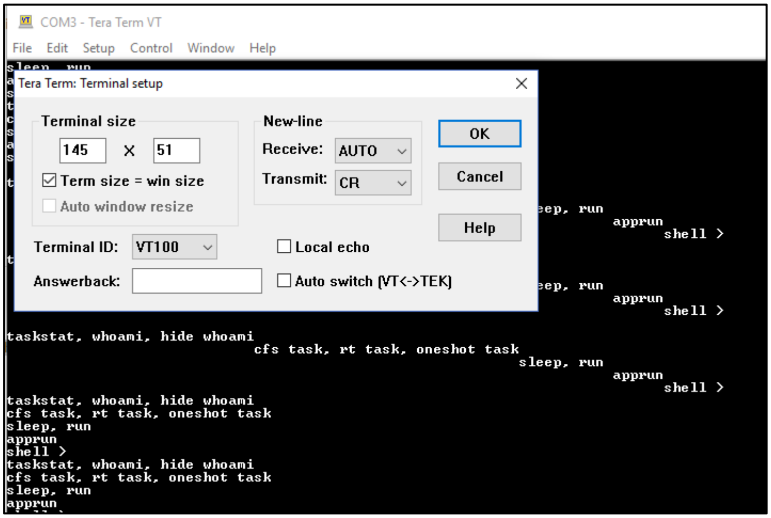
Port: COM5
Baud rate: 115200
Data: 8 bit
Parity: none
Stop: 1 bit
Flow control: none

Transmit delay
0 msec/char 0 msec/line

Figure 2-3 Teraterm serial port setting

You can also use a serial terminal application in your Ubuntu guest OS. First you have to forward the serial port from the Windows host OS to the Ubuntu guest OS. For this, in the Virtualbox window, select Devices -> USB and choose a serial to usb bridge as figure 2-5.

In Ubuntu guest OS, *Minicom* application can be used for UART serial communication with the target device and you can install it simply by '*sudo apt-get install minicom'* in your Ubuntu guest OS. Then start the Minicom with *'sudo minicom -s'*. Choose a 'Serial Device' by press shift + a and set Hardware Flow Control as 'No' by pressing shift + f**.** Settings of minicom terminal looks like figure 2-6. The serial device in the guest OS 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 as a default and run 'sudo minicom' without setting option from next time.


COM3 - Tera Term VT
File Edit Setup Control Window Help
Tera Term: Terminal setup
OK
Cancel
Help

Terminal size

145 × 51

☑ Term size = win size

□ Auto window resize

New-line

Receive: AUTO

Transmit: CR

Terminal ID: VT100

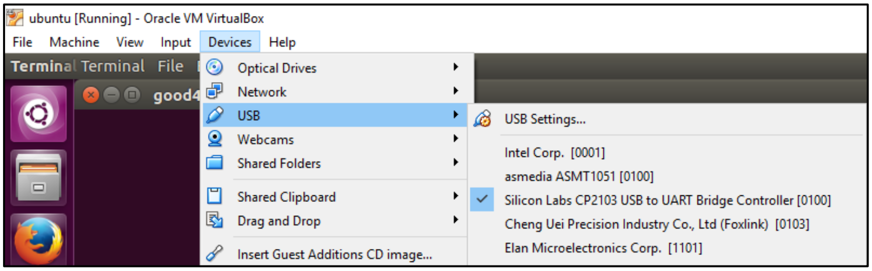
Answerback:

□ Local echo

□ Auto switch (VT<->TEK)

```
                                                                                    eep, run
                                                                                            apprun
                                                                                                  shell >
                                                                                    eep, run
                                                                                            apprun
                                                                                                  shell >
taskstat, whoami, hide whoami
                             cfs task, rt task, oneshot task
                                                            sleep, run
                                                                      apprun
                                                                            shell >
taskstat, whoami, hide whoami
cfs task, rt task, oneshot task
sleep, run
apprun
shell >
taskstat, whoami, hide whoami
cfs task, rt task, oneshot task
sleep, run
apprun
```

Figure 2-4: New-line setup in Teraterm



Intel Corp. [0001]

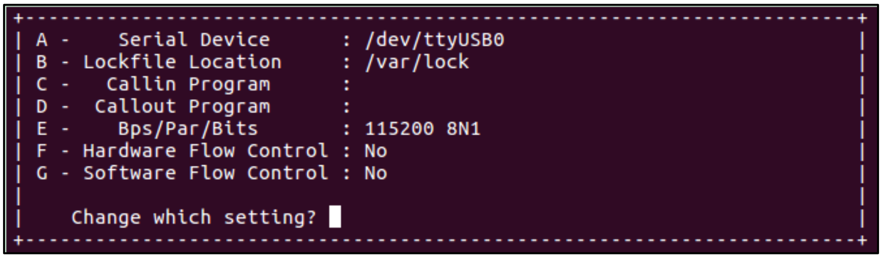
asmedia ASMT1051 [0100]

✓ Silicon Labs CP2103 USB to UART Bridge Controller [0100]

Cheng Uei Precision Industry Co., Ltd (Foxlink) [0103]

Elan Microelectronics Corp. [1101]

Figure 2-5 : USB to UART forwarding to guest OS



```
+-----------------------------------------------------------------------+
| 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 2-6 : Minicom serial port settings

2.3.4 GCC Tool Chain Installation

Now we have Ubuntu desktop working on Virtualbox. Next thing for the development PC setup is the installation of cross compiler into the Ubuntu. Keep in mind that SLOS will be running on ARM processor of PS subsystem in Zynq7000 chipset. Since the processor architecture of source compiling machine and binary running machine is different, we need a cross compiler which compiles the source files according to the target architecture. There are some commercial tool chains such as *Mentor Graphics' Sourcery CodeBench, ARM's RVDS,* but this book uses a *GCC (GNU Compiler Collection)* tool chain. You can download the GCC tools from ARM download site [9]. Currently, the SLOS is built by using 2017q1 GCC verion. You can also download it by cloning the tools repository as described in chapter 1.5. There is a GCC file named *'gcc-arm-none-eabi-6-2017-q1-update-linux.tar.bz2'* in *tools* repository. Decompress it with below command.

```
tar xvfj gcc-arm-none-eabi-6-2017-q1-update-linux.tar.bz2
```

Move it to *~/bin/* and rename it as *'*arm-2017q1'. Add the directory path of tool chain executables *(~/bin/arm-2017q1/bin)* to $PATH environment variable. To add the path, open *~/.bashrc* file and add the path to the PATH environment variable as below.

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

This also can be done by adding the path to the toolchain to the *Makefile*. Adding the toolchain path to the *Makefile* will be covered in chapter 3.7. But this book uses the toolchain path in the *bashrc* environment variable because we can use the GNU tools command anywhere in the shell by adding this path.

These steps including auto-build can be automatically done through Bitbake recipes. This is covered in Appendix A. Refer Appendix A if you are interested in Yocto or Bitbake build.

2.3.5 Xilinx Vivado, SDK Installation

Developers can program a custom hardware on PL subsystem by using Vivado IDE. In addition, we can design our custom hardware and develop its device driver. This book demonstrates how to define our own hardware by programming the PL subsystem. The process for custom hardware design and device driver development are described in chpater 7. Creating a Petalinux BSP for this customized hardware is quite automated by using the Vivado IDE. By using exported hardware configuration for both PS subsystem and PL subsystem, the SDK automatically creates the BSP for the FSBL (First Stage BootLoader). The Linux BSP is also created automatically by Petalinux tools. Until chpater 7 custom hardware, we will use a Xilinx bare Petalinux BSP.

Xilinx SDK also has useful tools like *XSDB* debugger. The XSDB debugger is necessary to connect the gdb server to the gdb client which is used for the debugging from the development PC.

To install Vivado, first download the Windows version of Vivado and SDK 2019.2, webpack edition from [7] and [8]. Vivado and SDK of Linux version are working very well, but this book assumes those tools are running in Windows OS. Webpack edition of Vivado is free and you can install the free license. Up until 2017, Xilinx has been releasing those tools version every quater of the year. After 2017, they release new versions two or three times per year. .

2.3.6 Petalinux Installation

The PS subsystem itself is a standalone system with ARM Cortex-A9 dual core. Xilinx tools create the bootloader and Linux binary image running on this PS subsystem. These tools have automated ways to create the Linux BSP and to build the images.

Petalinux must be installed in the Ubuntu guest OS in Virtualbox. For installing Petalinux properly, the Ubuntu must have additional packages installed. You can do this by running below command.

sudo apt-get install tofrodos gawk xvfb libncurses5-dev tftpd zlib1g-dev libssl-dev flex bison wget chrpath socat autoconf libtool texinfo libsdl1.2-dev gcc-multilib zlib1g:i386

After installing those packages, download Petalinux 2019.2 installer from [7]. The Petalinux version must be same as the Vivado and SDK version. Then install petalinux by running

./ petalinux-v2019.2-final-installer.run ~/${your_workspace}/petalinux-v2019.2

It takes a while to install Petalinux. Petalinux tool is necessary for building the final BOOT.BIN image which contains bootloader, FPGA bitstream and the custom OS. Petalinux also can create the default BOOT.BIN which is used for testing the Zynq7000 evaluation board. We will use Vivado IDE, SDK, and Petalinux through this book to build BOOT.BIN which contains SLOS kernel binary, PL subsystem bitstream.

2.3.7 Build Images for Default Block Design

In this chapter, we are going to follow the whole process of creating booting images for Zynq7000. This uses the default hardware configurations, bootloader and Petalinux without any customizations. In this chapter, we don't program custom hardware in the PL subsystem and we don't apply any modifications into the default flow in this demonstration.

Follow below steps to generate bare binaries for Zynq7000 evaluation kit. Bare binaries include petalinux image (image.ub) and BOOT.BIN image (design_1_wrapper.bit, FSBL, uboot.elf). We need to know how to build these two images because we will tweak those steps to build our SLOS image. We don't build FSBL and uboot images. We just copy them from tools repository described in chapter 1.5.2. If you are familar with Xilinx image build process, you can skip this chapter.

1) *Steps for FPGA bitstream generation*
   a) Run Vivado -> File -> Project -> New. Then click Next button until 'Default Part'

window. In Default Part window, click Boards tab and select ZYNQ-7 ZC702 Evaluation Board. Click Next -> Finish.

b) Go to Flow Navigator -> Create Block Design -> Click OK with default name and directory.

c) In the Diagram window, click add IP (+ button) and type 'zynq'. Then, double click 'ZYNQ7 Processing System' and click 'Run Block Automation'. Let everything default values and click OK button.

d) Click mouse left button on FCLK_CLK0 pin. Then drag the mouse to M_AXI_GP0_ACLK. This action will connect those two pins. You should see the diagram as in figure 2-7.

e) In Sources -> Hierarchy tab, click right mouse on design_1 (design_1.bd) and select 'Create HDL Wrapper' as figure 2-8. Select 'Let Vivado manage wrapper and auto-update' and click OK.

f) Now block designis done. Click 'Generate Bitstream' in the Flow Navigator to generate bitstream for PL subsystem.

g) After a while, bitstream generation will be done. The bitstream file for PL subsystem is placed in *${project_name}.runs/impl_1*. The file name must be design_1_wrapper.bit.

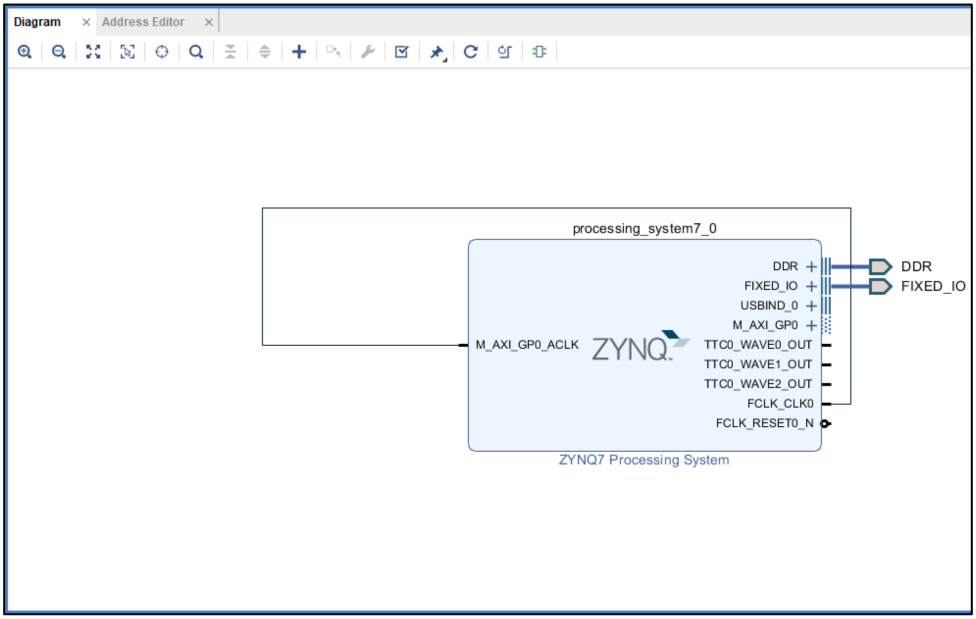


Figure 2-7: ZYNQ7 Processing System Block Design

2) *Steps for image.ub and BOOT.BIN image generation*

a) Export the block design in Vivado. Click File -> Export -> Export Hardware. Check ‘Include bitstream’ and click OK. Then, there is a new file named design_1_wrapper.hdf in *${project_name}.sdk* folder. Note the path of this file.

b) Go to the Ubuntu Virtualbox and press ctrl + alt + t button. This will run a command terminal window. Then go to Petalinux installation directory in the command terminal window. The directory name is normally *petalinux-v2019.2*. Run *‘*source settings.sh’ in installation root directory.

c) Now, let’s create a petalinux BSP for the default HW block that was just created in the previous step. Run below command.

```
petalinux-create --type project --name zynqdef --template zynq
```

This command will create a *zynqdef* folder for Zynq7000 chipset.

d) Go to *zynqdef* directory. Then, run

```
petalinux-config --get-hw-description=${path_to_hdf_file}
```

The file path noted in step 1) is used for the hdf file path. When configuration screen pops up, just press OK and EXIT. Xilinx petalinux tool automatically creates a BSP for the exported hardware. Connecting C or D drive in Host OS (Windows) to Guest OS is described in next chapter 2.3.8 for your convenience.

e) Now, Petalinux BSP is created. Build image.ub with a command below.

```
petalinux-build -v
```

After a while, all built images are placed in *image/linux* folder including petalinux image file (image.ub).

f) Build a BOOT.BIN image for booting the Zynq7000 evaluation kit. Run below command.

```
petalinux-package --boot --fpga design_1_wrapper.bit --fsbl zynq_fsbl.elf --u-boot=uboot.elf –force
```

As this command shows, the zynq_fsbl.elf, design_1_wrapper.bit and u-boot.elf are combined into BOOT.BIN.

g) Copy image.ub and BOOT.BIN file to SD card then try to boot the evaluation board as described in chapter 2.4.

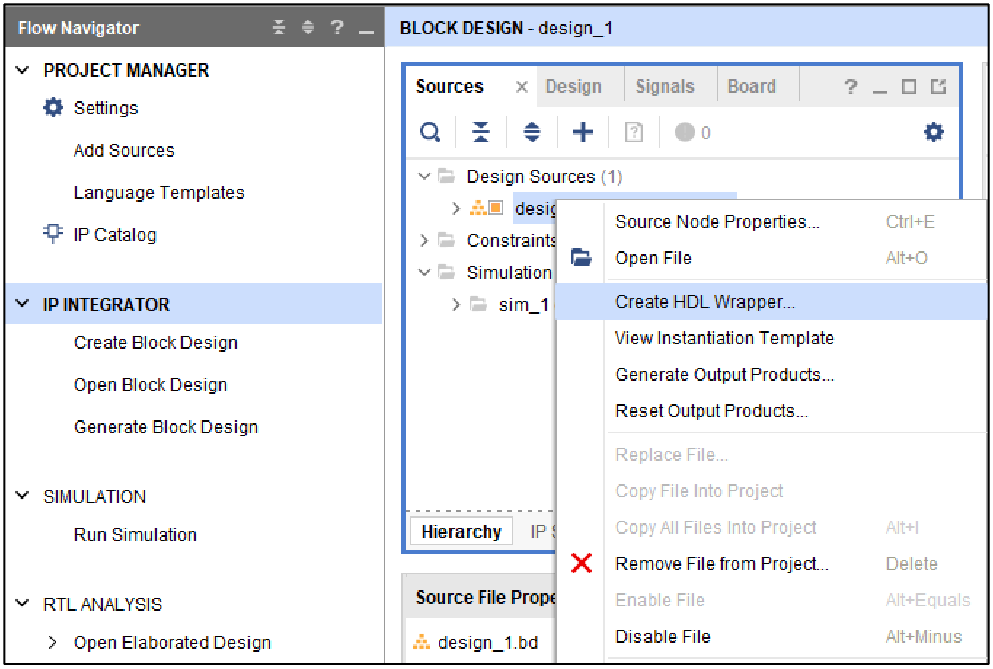


Figure 2-8: Create HDL Wrapper as a Top View of Block Design

2.3.8 Miscellaneous Settings

If finishing installation of ubuntu, let's do some fancy and convenient settings. Ubuntu needs 'sudo' keywords for running a command with a 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 asks a root passwd everytime and it seems very inconvenient to us, at least to me. Let's remove this annoying step. First run 'sudo visudo' and add '*${Your account} ALL=(ALL:ALL) NOPASSWD: ALL'* into the last 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 shell binary. Our development environment needs a bash as a default shell. Then, run 'sudo dpkg-reconfigure dash', and choose 'NO' in the question screen.

After Ubuntu16.04 installation, you have to install guest addition. Select Guest Addition image as in figure 2.9, and enter passwd , then guest addition will be installed.

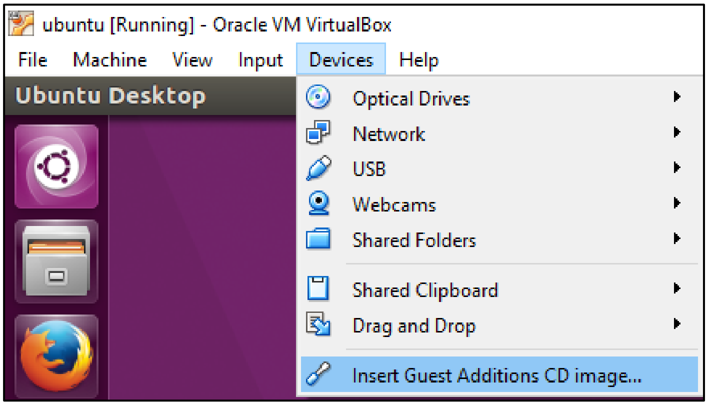


Figure 2-9 : Run guest addition in Ubuntu guest OS

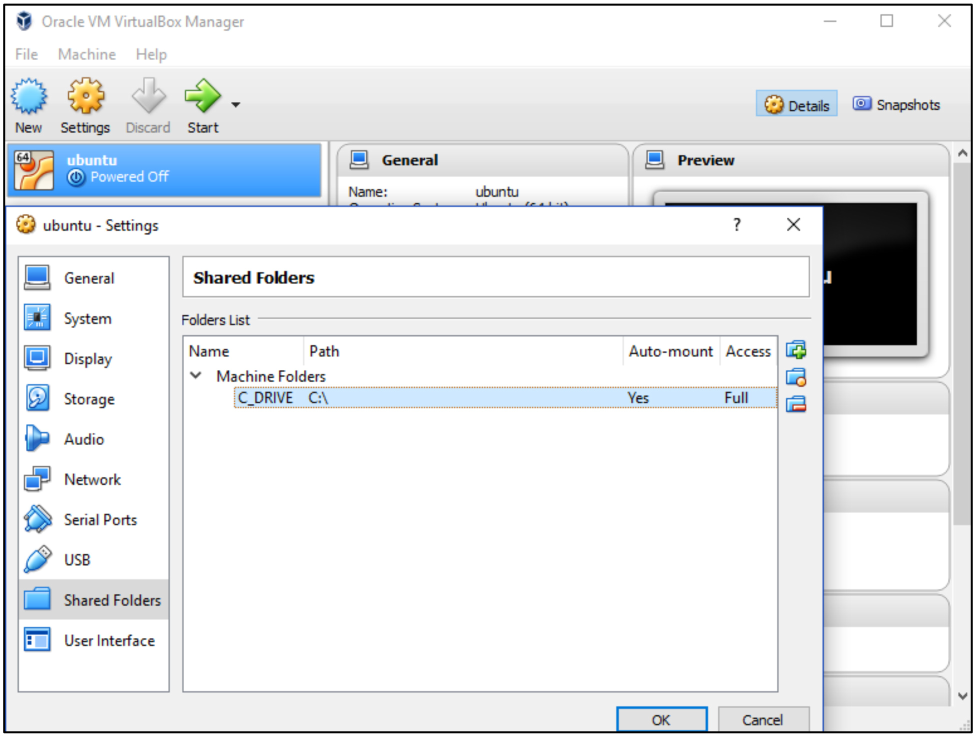
Next, let's set a folder to share files between host OS and guest OS. This setting is necessary for chapter 2.3.7 linking the bitstream hdf file to Petalinux BSP creation. Select the 'Shared Folders' in Virtualbox's Settings window as in figure 2-10. Choose the whole directory, for example C: drive, 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 member by running

```
sudo adduser ${your_id} vboxsf
```

This command gives you a permission to access the shared folder from guest OS. Now, you need to make a symbolic link to the shared folder. Run

```
ln -s /media/sf_C_DRIVE ~/c
```

Now, you can access the Host OS's shared folder by simply running *'cd ~/c'*. This is very convenient when you work in Virtualbox guest OS. If you want to link another disk like D drive, you can also link this drive with '~/d' symbolic link. We'll make our development code in the Host OS's disk storage and access it, build it from guest OS with using these links.



Shared Folders

Folders List

Name	Path	Auto-mount	Access
Machine Folders			
C_DRIVE	C:\	Yes	Full

Figure 2-10 : Setting shared folder between Host OS and Guest OS

2.4 Development Board Setup and Booting Petalinux

Xilinx has a couple of evaluation kits and this book uses ZC702 Evaluation Board for developing the custom operating system. This board has Zynq7000 CPU which has dual core ARM Cortex-A9 for its PS subsystem and Xilinx Artix-7 FPGA for its PL subsystem. It also has 1G DDR3 memory, USB, Ethernet PHY, USB-to-UART bridge, status LEDs, GPIOs for PS subsystem. You can refer [2] and [6] for its detailed features. This evaluation board seems to have too much for developing a simple operating system, but it has also a well-defined development, debugging tools which is essential for this type of software development.

All we need to know about the setup of the evaluation board is to set the boot mode. Zynq-7000 has following boot modes.

1) *Master boot mode (boot from flash)*
   a) Quad-SPI with optional Execute-in-Place mode
   b) SD Memory Card
   c) NAND
   d) NOR with optional Execute-in-Place mode,
2) *Slave boot mode*

a) JTAG boot mode

We will use SD Memory Card for booting Zynq7000 and need to set the SW16 dip switch for SD card boot mode. Following is SW16 configurations for each boot mode.

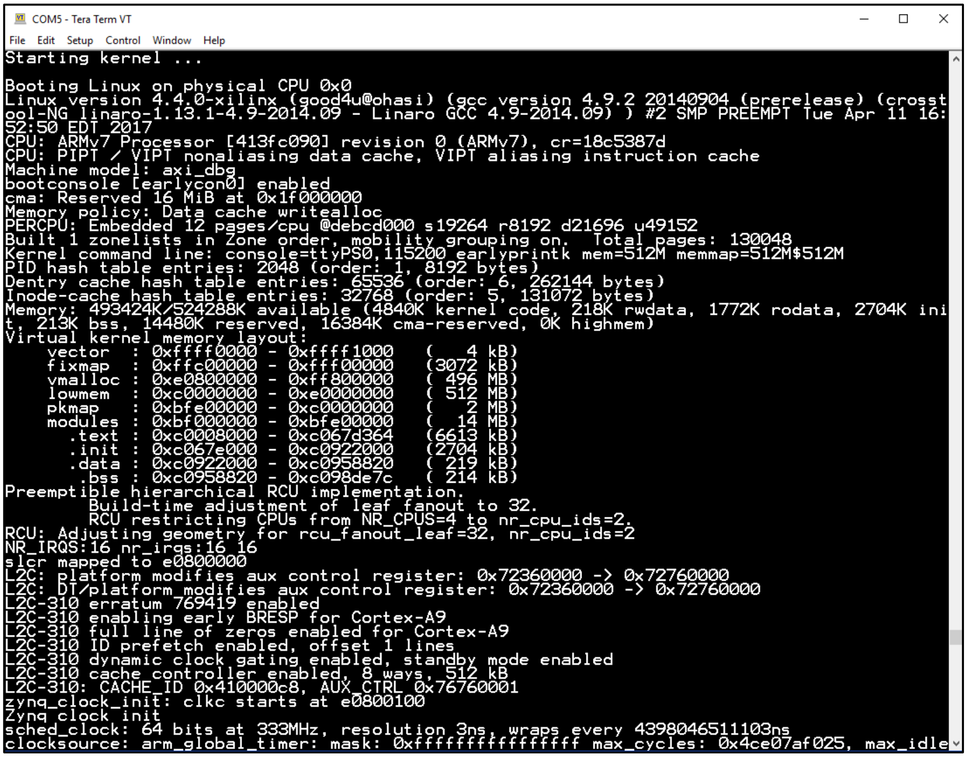
Boot Mode	SW16.1	SW16.2	SW16.3	SW16.4	SW16.5
JTAG mode	0	0	0	0	0
Independent JTAG mode	1	0	0	0	0
Quad SPI mode	0	0	0	1	0
SD mode	0	0	1	1	0
MIO Configuration pin	MIO2	MIO3	MIO4	MIO5	MIO6

Table 2-1: Zynq7000 evaluation kit boot mode

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 and is processed properly while the Zynq chipset starts to boot up.

As depicted in figure 2-1, the developement PC and Zynq evaluation board are connected through two USB cables. One is used for serial communication with the serial terminal in the host that we set up in chapter 2.3.3. The other USB connection is used to *XSDB (Xilinx System Debugger)* debugger. How to set up the XSDB debugger and how to use it is described in chapter 3.8. Just connect those USB cables properly for now.

Now, we have set up the developement environment, built the bitstream, bootloader, and petalinux images. It's time to boot up the board. Copy the images (BOOT.BIN, image.ub) files created in chapter 2.3.7 into SD card. Since we linked the Host Windows disk drive to Ubuntu Guest OS, you can easily copy them to SD card from Petalinux directory. Insert the SD card into the Zynq7000 evaluation board and turn on the board. If you don't follow the default image build steps in chapter 2.3.7, you can clone the tools repository in chapter 1.5.2. There are BOOT.BIN, image.ub files for Zynq7000. Copy those files to SD card and insert it to Zynq evaluation board. After power-on the board, you can see the booting messages in your serial terminal (teraterm or minicom) if you correctly setup the serial terminal as in chapter 2.3.3. This booting is for the default hardware configuration that we defined in previous chapters. And it also uses the Petalinux for its operating system. After the board boots up, you can run Linux shell commands, for example *'ls'*. This is because the image.ub file has an *init ramdisk* that has convenient Linux executables such as *busybox*.



```
Starting kernel ...

Booting Linux on physical CPU 0x0
Linux version 4.4.0-xilinx (good4u@ohasi) (gcc version 4.9.2 20140904 (prerelease) (crosst
ool-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 ini
t, 213K bss, 14480K reserved, 16384K cma-reserved, 0K highmem)
Virtual kernel memory layout:
    vector  : 0xffff0000 - 0xffff1000   (   4 kB)
    fixmap  : 0xffc00000 - 0xfff00000   (3072 kB)
    vmalloc : 0xe0800000 - 0xff800000   ( 496 MB)
    lowmem  : 0xc0000000 - 0xe0000000   ( 512 MB)
    pkmap   : 0xbfe00000 - 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: clkc 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
```

Figure 2-11: Linux booting message in Zynq evaluation board

2.5 ARMv7 Architecture Introduction

ARM is a *RISC (Reduced Instruction Set Computing)* processor architecture family designed by ARM holdings. They develop the architectures and license it to other companies (Samsung, Apple, Qualcomm and so on), who design their own products that implements one of those architectures. RISC processor has fixed length of instructions and consumes lesser power compared to *CISC (Complex Instruction Set Computing).* These characteristics are desirable for light, portable, battery-powered devices such as smartphones and embedded systems. The ARM architecture was first introduced in 1985 and still continue evolving significantly since its introduction. Eight major versions of the architecture have been defined up to date, denoted by the version numbers ARMv1 to ARMv8. Of these, the first three versions are now obsolete. ARMv3 to ARMv7 support 32bit instruction length, 32bit address space (Nonetheless, they still support 16bit instruction *Thumb mode,* which results in better code density). The ARMv8, released in 2011, is a 64bit architecture. Currently, ARM server computer is being develoed by using the ARMv8 architecture. The Zynq7000 chipset embedded in the evalution board has the Cortex-A9 dual core processor which is based on ARMv7 architecture.

The job of process management of operating system is to serve the processor hardware to other software applications. We have to understand the basic processor architecture before working on the real job of process management. It is the first step in operating system development to understand the processor hardware on which the operating system will be running. This book is not about the ARM processor or Zynq7000 processor, but will cover the least feature set of them needed for operating system development. For a detailed information of ARMv7 architecture, refer document [10].

2.5.1 ARM Processor Profile

ARM architectures have a set of variants. It has architecture profiles for different processor usage, *ISA (Instruction Set Architecture)* extensions and architecture extensions. Knowing these variants is necessary to understand the ARMv7 processor products. ARMv7 provides below three profiles:

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

2) *ARMv7-R*
   a) This profile is a *Real-time* profile, described as below.
   b) 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.
   c) Supports the ARM and Thumb instruction sets.

3) *ARMv7-M*
   Microcontroller profile, described as below.
   a) 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.
   b) Implements a variant of the ARMv7 PMSA. Supports a variant of the Thumb instruction set.

2.5.2 Architecture Extension

On top of basic ARMv7 architecture, ARMv7 has several extensions in instructions and architecture. Instruction Set Architecture extensions are:

1) *Jazelle*

   This extension 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.

2) *ThumbEE (Thumb Execution Environment)*

   This 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) A required extension to the ARMv7-A profile.
   b) An optional extension to the ARMv7-R profile.

   ARMv7-A must continue to include ThumbEE support, for backwards compatibility.

3) *Floating-point*

   This 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 several versions of the Floating-point (VFP) Extensions.

4) *Advanced SIMD*

   This 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.

Besides the instruction set extension, there are architecture extensions. The architecture extensions are:

1) *Security Extensions*

   These 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.

2) *Multiprocessing Extensions*

   These extensions are an optional set of extensions to the ARMv7-A and ARMv7-R profiles, that provides a set of features that enhance the multiprocessing functionality.

3) *Large Physical Address Extension*
This 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 (LPAE)* requires the implementation of Multiprocessing Extensions.

4) *Virtualization Extensions*
These 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.

5) *Generic Timer Extension*
This is an optional extension to any ARMv7-A or ARMv7-R, that provides a system timer, and a low-latency register interface to it.

6) *Performance Monitors Extension*
This extension is not part of the architecture before ARMv7. ARMv7 architecture manual [10] was the first architectural specification of this extension. The basic form of performance monitor is a cycle counter, an event counter and controls for these counters.

Some of these extensions are applied to ARM Cortex-A9 architecture. This will be covered in chapter 2.6.

2.5.3 ARM ISA Overview

ARMv7 Architecture defines its Instruction Set Architecture (ISA). ARM ISA is a *load/store architecture* that must load instruction before arithmetic operations and store the result into the memory again. So, a simple arithmetic calculation is divided into two phases: a memory access which loads or stores between memory and register, and ALU operations which occur between registers. The example processor of load/store architecture are RISC chipsets such as ARM, MIPS, PowerPC and so on. There is other ISA architecture such as *register memory architecture*. The register memory architecture can run an arithmetic operation between memory and register. In this ISA architecture, one of the ADD operand can be in memory and the other operand is in register. The CISC instruction set such as x86 is an example of register memory architecture.

Then, most ARMv7 instructions are categorized as one of *Data Processing Instructions*, *Load/Store Instructions* or *Branch Instructions*.

1) *Data Processing Instructions*

   This type of instruction is used for processing data in register such as add, subtract, shift, bit operations, and logical operations. These operations are done in *ALU (Arithmetic Logic Unit)*. Before processing data, ARM first needs to load the operand's data from memory to a register.

2) Load/Store Instructions

   These instructions are 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 and exclusive load/store for synchronized access to the memory. Load/Store instruction examples are *ldr, str, ldm, stm, ldrex, strex*.

3) Branch Instructions

   These instructions are used for branch to target address such as *B, BL, BLX*.

ARM Technical Reference Manual [10] chapter A8.8 describes the ARM assembler instructions in an alphabetical order. SLOS needs a little bit of ARM assembler programming. The assembler programming is used for programming for exception handlers, context switching, coprocessor registers for MMU programming, and so on. There are over 400 ISA instructions described in that chapter. You can refer the chapter A8.8 while reading ARM assembler code.

Data processing instructions are quite self explanatory; the operations of addition, subtraction, shift instructions are very clear. But there is also some ARM specific information to understand the load/store instruction. We have to know how ARM ISA defines its load/store command in detail to implement our own exception vectors, context switching, and MMU manipulation.

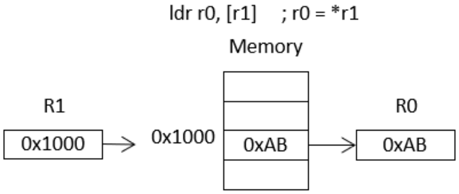


Figure 2-12: An example of simple load instruction

In figure 2-12 shows the simplest load instruction to load a data in memory address r1 to a register r0. The bracket mark ([ ]) is used to access to a specific memory address. In figure 2-12, register r1 value is 0x1000, then [r1] in assembler codes means the data in memory address 0x1000. This is same as pointer operation in C programming language. The [r1] in ARM assembler is same as *r1 in C programming language in the figure 2-12 example.

There are two load/store address indexing modes in ARM.

1) *Pre-indexing*

   The memory address is updated first and load/store the data of that memory address to/from the register. This address indexing mode is depicted in figure 2-13. The *'ldr r0, [r1, #4]'* instruction means load the data in memory [r1 + 4] address. In below example, the pre-indexed load/store operations happen in memory address 0x1004 which is 4 bytes increased from the value in register r1.

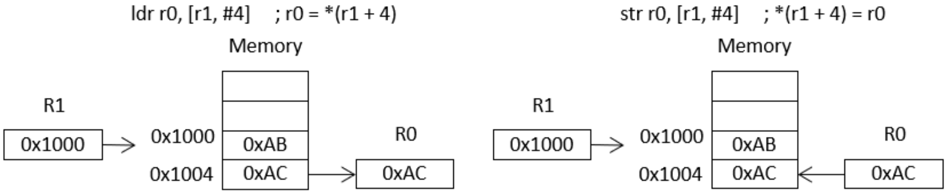


Figure 2-13: Pre-indexing of load/store instruction

2) *Post-indexing*

   Post-indexing, as the name means, is the opposite mode to the Pre-indexing mode. The data in the memory is loaded/stored to/from register and the second operand is updated. This is depicted in figure 2-14. The load/store operations happen in memory address 0x1000 which is the value of register r1, then the value of register r1 is increased by 4 bytes. The final value of register r1 is 0x1004.

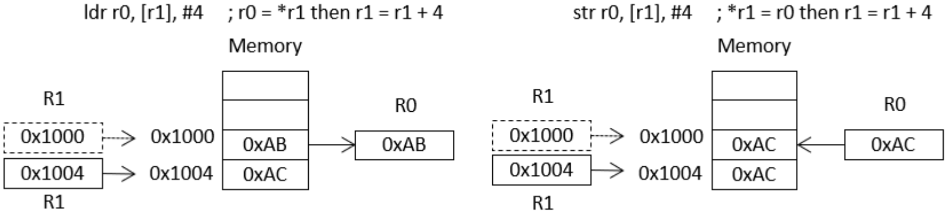


Figure 2-14: Post-indexing of load/store instruction

There is a variation with *'!'* mark which represents a write-back to the second operand register. For example, *ldr r0, [r1, r2]!* assembler code means r0 = *(r1 + r2) and then r1 = r1 + r2. This write-back operation is frequently used with stack pointer.

Another thing to know about load/store is load multiple and store multiple instructions for multiple load/store operations at a time. These instructions can load/store a memory block at one instruction. Of course, this operation is better for performance. The multiple load/store instruction have four operation modes.

1) *IA*
   Increment After. Increment address After each transfer
2) *IB*
   Increment Before. Increment address Before each transfer
3) *DA*
   Decrement After. Decrement address After each transfer
4) *DB*
   Decrement Before. Decrement address Before each transfer

With these 4 operation modes, there are load/store instruction variations such as *ldmia, ldmib, ldmda, ldmdb, stmia, stmib, stmda and stmdb* instructions. These instructions are covered in detail in chapter 2.5.5 with stack pointer register.

One interesting feature in ARM instruction set is that all ARM instructions support conditional execution. Table 2-2 summarizes condition code and its suffix used in ARM assembler instruction. Following is an example for the usage of condition flag suffix. Let's consider a simple C code as below.

```
for (i = 10; i != 0; i--) {
        do_something();
}
```

This routine can be programmed as ARM assembler like below. The line 6, branch instruction, branches to LOOP if the condition 'Not Equal' is met. The original branch instruction is extended with the conditional suffix. The condition flag (Z flag) is set after the *cmp* instruction. The condition flag is covered in chapter 2.5.5.

```
1   mov     r4, #10
2   LOOP:
3   bl      do_something
4   sub     r4, r4, #1
5   cmp     r4, #0
6   bne     LOOP
```

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

Table 2-2: ARM assembler condition code

2.5.4 ARM Processor Modes

ARMv7 architecture reference defines several ARM processor operation modes. ARM processor modes are same as the *exception* state while program running. Basically, there are 7 operation modes plus *Monitor mode* for security extension and *Hypervisor mode* for virtualization extension. ARMv7 processor exception modes are as follows.

1) *User (USR) Mode*

   This mode is non-privileged mode limited access to system hardware resources. This mode is used for executing most application programs. CPSR mode bit value is 0x10.

2) *Fast Interrupt (FIQ) Mode*

   This mode is a privileged mode used for handling fast interrupts. CPSR mode bit value is 0x11.

3) *Interrupt (IRQ) Mode*

   This mode is a privileged mode used for general-purpose interrupt handling. CPSR mode bit value is 0x12.

4) *Supervisor (SVC) Mode*

This mode is a privileged mode used for the operating system. CPSR mode bit value is 0x13.

5) *Abort (ABT) Mode*

   This mode is entered after a data abort or prefetch abort. CPSR mode bit value is 0x17.

6) *System (SYS) Mode*

   This mode is a privileged mode used for the operating system. CPSR mode bit value is 0x1f.

7) *Undefined (UND) Mode*

   This mode is entered when an Undefined Instruction exception occurs. CPSR mode bit is 0x1b.

8) *Monitor (MON) Mode*

   This mode is a Secure mode for the Security Extensions to run Secure Monitor code. CPSR mode bit is 0x16.

9) *Hypervisor (Hyp) Mode*

   This mode is a Hypervisor mode for Virtualization Extensions to handle the exceptions in virtual operating system.

Modes other than User mode are collectively known as *privileged modes.* Privileged modes are used to service interrupts, exceptions, or to access protected resources such as interrupt enable/disable bits in *CPSR* register. All of these modes are not used in SLOS. The modes used in SLOS are IRQ mode, SVC mode, and ABT mode. Monitor mode, Hypervisor mode are only applied to very high-level operating system like Linux. In this chapter, it is enough to note that ARM processor has modes for its special usage and some of them has privileged rights to access hardware but some of them doesn't have.

2.5.5 ARM Core Registers and Program Status Registers (PSRs)

ARM registers are grouped into ARM core registers and special purpose registers. Special purpose registers are *System Control Registers (SCTLR), Secure Configuration Registers (SCR), Coprocessor Registers (CP), Program Status Registers (PSR)* and so on. In this chapter, we look into ARM core registers and program status registers and how they are used in SLOS development. As mentioned, ARM core registers are composed of as below.

1) Thirteen general purpose registers: R0 to R12 that software can use for its logical

operations.

2) SP, the stack pointer, that can also be referred to as R13.
3) LR, the link registrer, that can also be referred to as R14.
4) 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 along with load/store instructions. Stack is a small memory region which is used for the ARM processor to store temporary data while running a program such as subroutine function calls, or function's local variable storage. We don't much care about the stack pointer operations when working on high level application programming but in operating system development, we need to program a stack memory for each ARM mode and for each forked task. Knowing stack is important for custom OS development and let's look into it in more detail. In addition to load/store operation described in chapter 2.5.3, ARM stack can have 4 more operation modes *(Full/Empty, Ascending/Descending).* So, all operation modes that stack can have are as below.

1) *IA*
Increment After. Increment address After each transfer.

2) *IB*
Increment Before. Increment address Before each transfer.

3) *DA*
Decrement After. Decrement address After each transfer.

4) *DB*
Decrement Before. Decrement address Before each transfer.

5) *FD*
Full Descending stack. Address decreases in store command and increases in load command. Current stack pointer (SP) has an address for a valid data (full). Operation is same as Decrement Before (DB) in store command and same as Increment After (IA) in load command.

6) *ED*
Empty Descending stack. Address decreases in store command and increases in load command. Current stack pointer points to an address that has not yet loaded data (empty). Operation is same as Decrement After (DA) in store command and same as Increment Before (IB) in load command.

7) *FA*
Full Ascending stack. Address increases in store command and decreases in load command. Current stack pointer points to an address with a valid data (full). Operation is same as Increment Before (IB) in store command and same as

Decrement After (DA) in load command.

8) *EA*
Empty Ascending stack. Address increases in store command and decreases in load command. Current stack points to an address that has not yet loaded data (empty). Operation is same as Increment After (IA) in store command and same as Decrement Before (DB) in load command.

If address is increased in each transfer, stack is in *Ascending mode*. If address is decreased, *Descending mode* stack is used. Also, there are two more cases *(empty or full)* based on the time of address change before or after load/store transfer. There are other stack operation instructions; *push, pop*. *'push'* instruction pushes the value in registers onto, and *'pop'* instruction pops value in registers off a full descending stack. SLOS uses *full-descending mode* for its exception mode stack and for all tasks' stack. Figure 2-15 through figure 2-18 illustrate the stack operation modes and corresponding ARM instructions.

The LR register holds the return address. PC register has the address of current running instruction plus 8 while executing ARM instructions. This is because ARM has a pipeline of instructions and the address of fetched instruction (PC address) is current executing instruction plus 8.

R0 to R12 are used for general purpose. But especially, when execution flow jumps to other function, there is a procedure call standard defining how to use these registers. The *ARM Architecture Procedure Call Standard (AAPCS)* is part of the ABI (although the ABI actually calls it the Procedure Call Standard for the ARM Architecture). It specifies conventions for register and stack usage by the compiler and during subroutine calls. Knowledge of this is vital for inter-working C and assembly code and can be useful for writing optimal code. The AAPCS supersedes the previous ARM-Thumb Procedure Call Standard.

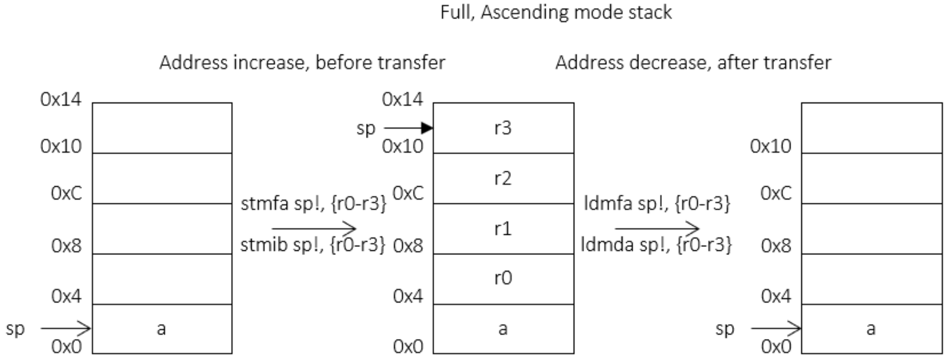


Figure 2-15: An example of load/store multiple in full, ascending stack. Store instruction is same as Increment Before (IB) and Load instruction is same as Decrement After (DA).

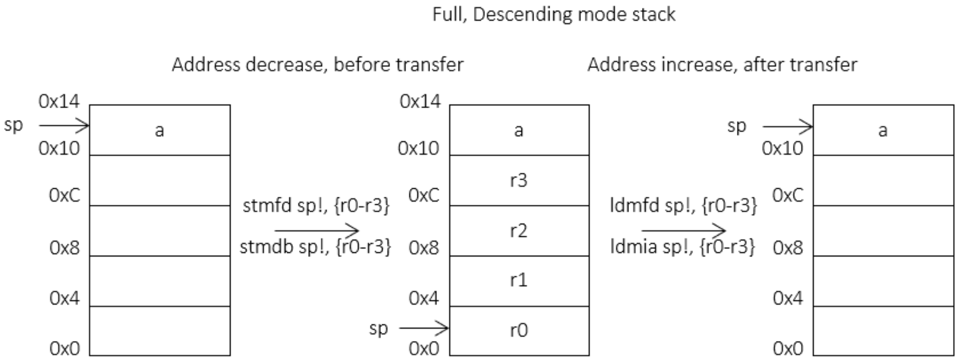


Figure 2-16: An example of load/store multiple in full, descending stack. Store instruction is same as Decrement Before (DB) and Load instruction is same as Increment After (IA).

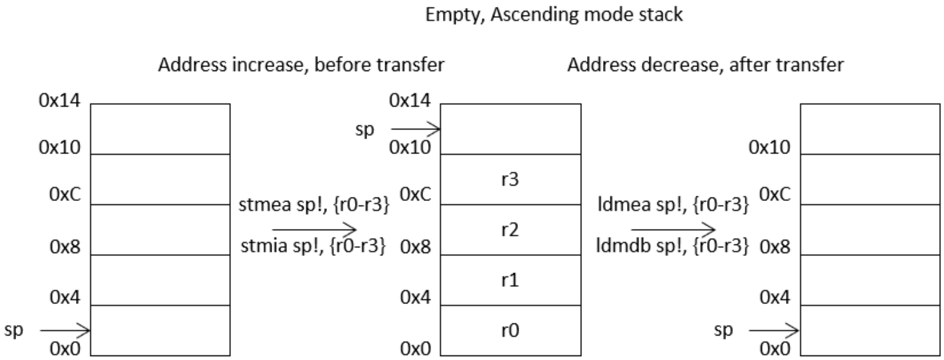


Figure 2-17: An example of load/store multiple in emtpy, ascending stack. Store instruction is same as Increment After (IA) and Load instruction is same as Decrement Before (DB).

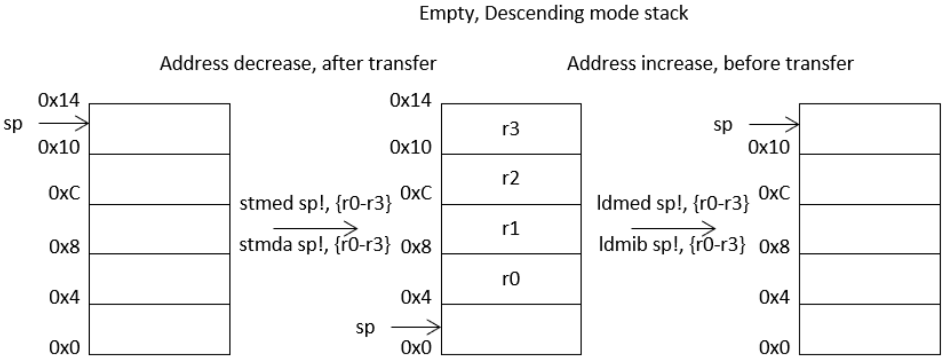


Figure 2-18: An example of load/store multiple in empty, descending stack. Store instruction is same as Decrement After (DA) and Load instruction is same as Increment Before (IB).

The AAPCS specifies rules that must be adhered to by callers to enable a callee function to run and what callee routines must do in order to ensure that callers can continue function correctly when the callee returns. It describes the way that data is laid out in memory and how the stack is laid out, plus permitted variations for processor extensions. It defines how code that has been separately compiled or assembled works together. *Caller* function is for current running function and *Callee* function is a subroutine function called by 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. Following table 2-3 summarizes the registers in AAPCS.

Register	PCS name	PCS role
R0	a1	Caller saved registers. Argument / return value registers. These hold the first 4 function arguments on function call and return value on function return. A callee function can use these registers as general scratch registers.
R1	a2	
R2	a3	
R3	a4	
R4	v1	Callee saved registers. Registers for general local variables. The callee function must preserve the register values of caller function.
R5	v2	
R6	v3	
R7	v4	
R8	v5	
R9	tr/sb/v6	Static base / register variable.
R10	v7	Register variable.
R11	v8	Register variable.
R12	IP	A general scratch register that a function can corrupt.
R13	SP	Stack pointer. This always points at the top of the stack.
R14	LR	Link register.
R15	PC	Program counter.

Table 2-3: Summary of ARM core registers and its AAPCS description

R11 can be used as a *frame pointer*. Frame pointer points to the start address of a current function's stack. It is different with the 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 jumping to another function. Since frame pointer has the start location of current function's stack, we can use it to save function's meta information (frame information) such as pc, lr, sp, and fp. With this frame information, we can easily back trace the function call chains. In the below example, the callee function saves caller function's frame information to the start location of its stack and use the saved information when it returns to the caller function. Please notice that fp and sp information comes from caller function (previous function), and lr, pc values 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. Below is an example of how the frame record are stored to the stack when funcA is called from some other function.

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

Line 2~4 stores the frame record to stack and update the current function's frame pointer with the start address of its stack. When funcA returns, it restores the stack, frame pointer and pc value of the previous function (caller function) with the saved frame record.

R12 is used for general scratch register which can be corrupted between function calls.

R13 ~ R15 registers are architecture defined such as R13 for the stack pointer, R14 for link regsiter to save the return address and R15 for current program counter register. We can't change the usage of these registers for custom purpose.

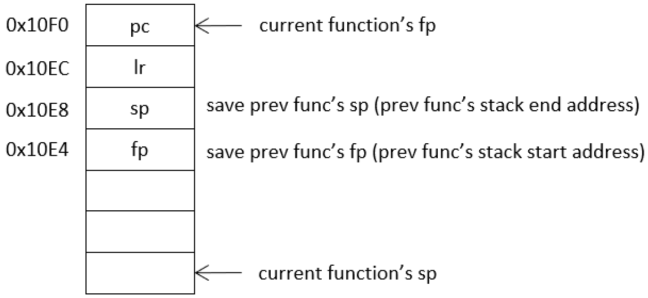

0x10F0
pc
current function's fp
0x10EC
lr
0x10E8
sp
save prev func's sp (prev func's stack end address)
0x10E4
fp
save prev func's fp (prev func's stack start address)
current function's sp

Figure 2-19: Frame record stored into stack

ARM also has many special purpose registers. Most of them are used for system level configuration such as MMU configuration, Secutiry configuration and so on. You can refer [10] for detailed information of them. But at least, SLOS development needs to manipulate one special purpose register, *Current Program Status Register (CPSR register). SPSR (Saved Program Status Register)* is a banked copy of CPSR register. CPSR register looks like figure 2-

20.

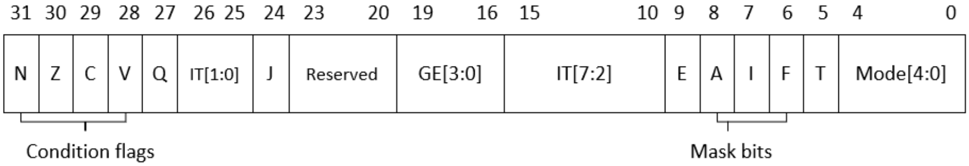


Figure 2-20: Description on CPSR register field

1) *Condition flags*
   Bits[31:28]. Set on the result of instruction exeution. The condition flag can be read or written in any processor mode.
   a) N, bit[31] is for Negative condition flag. If the result of instruction is negative, then the processor set N to 1.
   b) Z, bit[30] is for Zero condition flag. Set to 1 if the result of instruction is zero, and set to 0 otherwise.
   c) C, bit[29] is for Carry condition flag. Set to 1 if the result of instruction results in carry.
   d) V, bit[28] is for Overflow condition flag. Set to 1 if the result of instruction results in an overflow condition.

2) *Q, bit[27]*
   This bit is for Cumulative saturation bit.
3) *IT[7:0]*
   Bits[15:10, 26:25]. If-Then execution state bits for the Thumb IT(If-Then) instruction.
4) *J, bit[24]*
   This bit is for Jazelle bit
5) *Bits[23:20], Reserved*
6) *GE[3:0]*
   Bits[19:16]. Greater than or Equal flags for the parallel addition or subtraction (SIMD) instructions.
7) *E(ndianness)*
   Bit[9]. Endianness execution state bit. Controls the load/store endianness when access to the memory. 0 is for Little-endian operation, 1 is for Big-endian operation. Instruction fetches ignore this bit.
8) *Mask bits*
   Bits[8:6]. These mask bits are set only at privilege level 1 or higher. Their values can be read in privilege level 0 (unprivileged mode), but can be written only from level

1 or higher. Privilege level will be explained in chapter 2.5.7. While operation system development, we need to manipulate these mask bits.

a) A, bit[8] is for asynchronous abort mask bit. 0 means Asynchronous abort is enabled, 1 is not.
b) I, bit[7] is for IRQ mask bit. 0 means unmasked IRQ, that is, IRQ interrupt is enabled, 1 is not.
c) F, bit[6] is for FIQ mask bit . 0 means FIQ is enabled, 1 means FIQ is disabled.

9) *T(humb) bit*

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.

10) *Mode bit*

Bits[4:0]. This field determines the current mode of processor. Table 2-4 is the summary of mode bits and associated processor modes.

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	This mode is entered when executing a HVC (Hypervisor Call) instruction in Non-secure PL1 mode. This mode is a part of Virtualization Extension.
Monitor		0x16	MON	This mode is entered when executing an SMC (Secure Monitor Call) instruction in a PL1 mode. This mode is a part of Security Extension.

Table 2-4: Description on CPSR mode bits and processor modes

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. Below table describes the banked ARM core registers (Hyp mode and Monitor mode are not shown here, refer [10]).

User mode and System mode share all their registers and R8_usr ~ R14_usr registers are their own copies. They don't share the R8_usr ~ R14_usr 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 the supervisor mode loads its own stack when it is activated. SPSR register saves the CPSR register 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_USR register. As described in chapter 1, SLOS doesn't support USR, SYS modes. All tasks run in SVC mode of privileged level and there is no mode change in the processor.

User and System	Supervisor	Abort	Undefined	IRQ	FIQ
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8_USR	R8	R8	R8	R8	R8_FIQ
R9_USR	R9	R9	R9	R9	R9_FIQ
R10_USR	R10	R10	R10	R10	R10_FIQ
R11_USR	R11	R11	R11	R11	R11_FIQ
R12_USR	R12	R12	R12	R12	R12_FIQ
SP_USR	SP_SVC	SP_ABT	SP_UND	SP_IRQ	SP_FIQ
LR_USR	LR_SVC	LR_ABT	LR_UND	LR_IRQ	LR_FIQ
PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_SVC	SPSR_ABT	SPSR_UND	SPSR_IRQ	SPSR_FIQ

Table 2-5: Banked ARM core registers

2.5.6 Security States

ARMv7-A Technical Reference Manual [10] defines the Secure state and Non-secure state of the processor. The Secure state is also known as Secure world while the Non-secure state is also known as Normal world. In ARMv7-A architecture, the introduction of the *TrustZone* Security Extension creates two security states for all processor modes, except Mon mode and Hyp mode, as shown in blow table 2-6.

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

Table 2-3: Security state of processor modes

The MON mode runs only in Secure world and is the only channel that Non-secure world can use to communicate with the Secure world. There could be some confusions steming from these many different processor modes. Using Security or not is just application dependent. SLOS doesn't need high level of security and doesn't interchange between Secure world and Non-secure world. Because of the fact that when the process is reset, it goes to secure state by default, SLOS always runs in Secure world. Nonetheless, it doesn't secure at all; Secure world just one mode of ARM processor. Knowing current secure state can be done by checking the *SCR (Secure Configuration Register)* register's NS bit (bit[0]). If this bit is set, then current processor mode is Non-Secure Mode, if this is clear, processor mode is on Secure Mode.

One noticeable thing about the security state is that the processor's reset vector supports only Secure Mode, which means when SLOS jumps to reset vector, the processor is already in Secure Mode. If SLOS is to support interchange between Secure Mode and Non-Secure Mode, this reset vector needs to implement Monitor Mode and load non-secure operating system for non-secure world. The non-secure applications can access secure world only through the *SMC (Secure Monitor Call)* system call. The Implementation of interchanging between Secure mode and NonSecure mode is beyond the scope of this book. Let's keep staying in Simple and Light.

2.5.7 Privilege Levels

The ARMv7 architecture defines different levels of execution privileges.

1) Secure state has the privilege levels PL1 and PL0
2) Non-secure state has the privilege levels PL2, PL1, and PL0.

PL0 indicates unprivileged execution in the current security state. The current processor mode described in chapter 2.5.4 determines the execution privilege level, and therefore the execution privilege level is same as the processor privilege level. Every memory access also has an access privilege, that is either unprivileged or privileged. The characteristics of each privilege levels are as follows.

1) *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.

2) *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. Software executing at PL1 makes privileged memory accesses by default, but can also make unprivileged accesses.

3) *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.

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. As mentioned, the MON mode is the only channel between Secure Mode and Non-secure Mode. The fact that Non-secure Hyp mode executes at PL2 does not indicate that it is more privileged than the Secure PL1 modes. Secure PL1 modes can change the configuration and control settings for Non-secure operation in all modes, but Non-secure modes can never change the configuration and control settings for Secure operation.

2.5.8 Relationship among Mode, Privilege, and Security

Figure 2-21 summarizes the relationship among privilege levels, security mode and non-secuar mode. It shows that in the non-secure state, the processor has PL0 for User Mode, PL1 for Exception Modes, and PL2 for the Hypervisor Mode, while, in the Secure state, the processor has only PL0 and PL1 privilege levels, and plus Mon Mode at PL1. Each privilege level in different secure worlds has its own processor mode copies. For example, there are SVC mode in non-secure mode and same SVC mode in secure mode as well. But those modes are handled differently (different security level) in the processor.

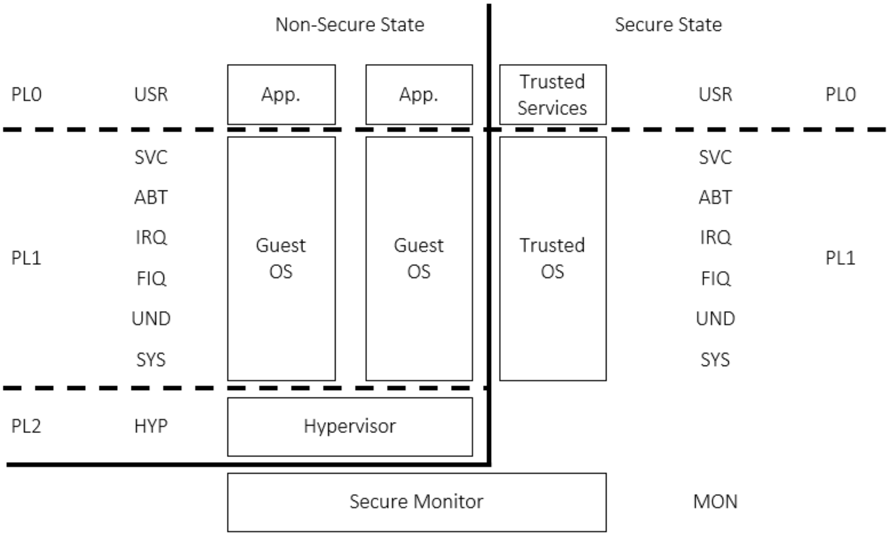


Figure 2-21: Relationship of Security States, Privileged Modes, and Processor Modes

Secure Monitor can access both Secure and non-secure state. But the application executing in non-secure state cannot use system registers associated with the secure state. For example, a Hypervisor executing at PL2 cannot access any of the secure system registers. Secure Monitor Mode does a context switch between the secure state and non-secure state. Secure Monitor also works as a gate keeper for the access from non-secure state to secure state. The Secure Monitor Mode can be entered by interrupt, external abort exceptions or explicitly by *SMC* system call from non-secure state. Secure state can easily go into Monitor mode simply by changing the CPSR register mode bit.

2.6 Cortex-A9 and Cortex-A9 MPCore Introduction

In addition to ARM architecture extension described in chapter 2.5.2, ARM has branch product by adding more features. 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 [11]. It expands the ARMv7 architecture by adding those features and defining the system-wide architecture.

Cortex-A9 MPCore expands again the Cortext-A9 processor. 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 multiple processors. Cortex-A9 MPCore also has a set of peripherals such as global timer, watchdog, private timer, Generic Interrupt Controller (GIC). Chip makers can compose their system with Cortex-A9 MPcore along with other their own IPs.

A detailed description on these processors is skipped here. Just remember that Cortex-A9, Cortex-A9 MPcore processors are a branch of ARMv7 and expands the design from ISA to a system-wide design by defining interfaces between processors, interrupt controller, SCU, timer and so on. Some of these features for SLOS implementation will be covered in each chapter if it is necessary.

2.7 Xilinx Zynq7000 Introduction

This chapter describes an overview of Xilinx Zynq7000 and ZC702 evalution kit which we are going to use to implement the SLOS. Xilinx Zynq7000 SoC family is a good chipset product to practice developing a custom operating system. Zynq7000 processor is manufactured by Xilinx which is packaged with dual or single-core ARM Cortex-A9 MPCore based processing subsystem *(PS Subsystem)* and Xilinx programmable logic subsystem *(PL Subsystem)* in a single chipset.

As described in previous chapter, the ARM Cortex-A9 MPCore is a 32-bit processor core licensed by ARM Holdings implementing the ARMv7-A architecture [12]. Processing subsystem has its own hardware resources such as I/O peripherals, memory interfaces, ethernet MAC and so on. It is a master boot processor which downloads the FPGA bitstream to the programmable logic subsystem during boot time. This PS subsystem can run a standalone application, commercial simple operating system such as *FreeRTOS*, or a high-level operating system such as Linux.

Unlike PS subsystem which is developed for general purpose software running, the PL subsystem can run a custom programmed hardware. This programmed hardware could be a *coprocessor* optimized for a specific operation such as image processing, or could be an I/O bound custom DMA processing, for example. User-programmed bitstream for programmable logic is downloaded during booting from PS subsystem.

Xilinx supports many prebuilt IPs in their repository and a user can simply add many of them to his hardware design for free. By developing a custom hardware in a programmable logic subsystem, a user can achieve a high performance, real time system for a 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 processed by a user application running in the PS subsystem. Moreover, the PL subsystem is a perfect real time system and the processing time in PL subsystem is highly deterministic if it achieves timing closure.

So, PS subsystem is used for the general-purpose applications like controlling the custom hardware in PL subsystem, network applications. A custome hardware programmed in PL subsystem is used for high performance, deterministic and time closed applications. How to partition the application logic into PS subsystem and PL subsystem is called hardware-software codesign. Zynq7000 is well suited for this purpose. SLOS will cover the hardware-software codesign by designing the cooperation between the PS subsystem and PL subsystem in chapter 7.

The communication between PS and PL is done through the AXI4 interface. AXI4 interface is developed by ARM Holdings as a part of ARM AMBA microcontroller bus architecture. We don't need to know the details of AXI interface in this operating system development but when we use Xilinx Vivado to design a 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 [1].

2.7.1 Zynq7000 PS Subsystem Blocks

Figure 2-22 illustrates the top-level view of Zynq7000 PS subsystem functional blocks [2]. The Zynq7000 chipset has following major functional blocks.

1) *Processing Subsystem (PS Subsystem)*
   a) Application Processing Unit (APU)
   b) Memory Interfaces
   c) I/O peripherals (IOP)
   d) Interconnect

2) *Programmable Logic (PL Subsystem)*

Xilinx composes the PS subsystem in Zynq7000 by integrating each sub-block from different vendors. For example, the Cortex-A9 MPCore comes from ARM, USB is supplied from Synopsys, Gigabit Ethernet MAC from Cadence, etc. PL subsystem is derived by Xilinx *Artix-7* or *Kintex-7* FPGA.

In figure 2-22, the PS subsystem is a standalone system which has its own processors, caches, interrupt controller, timer peripherals, DMA, IO peripherals (USB, GigE, SDIO, UART, GPIO, I2C), external memory interfaces. SLOS will run on the Cortex-A9 MPcore on this PS subsystem.

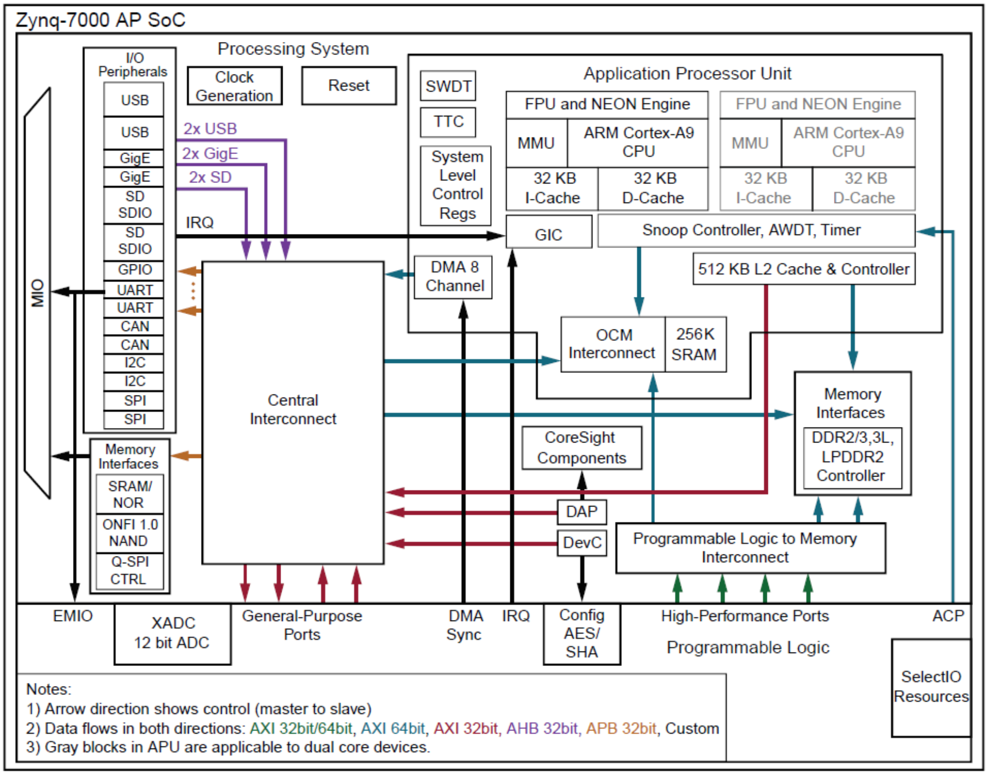


Figure 2-22: Top View of subblocks in PS subsystem from UG-585 Zynq7000 Technical Reference Manual

2.7.2 Zynq7000 Booting Sequence

The figure 2-23 summarizes the Zynq7000 booting sequence. It is composed of 3 booting stages after POR (Power on Reset). Those are 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 in developing SLOS because the SLOS executable will be inserted into the stage 2 by replacing the UBoot binary with SLOS binary. Stage 2 booting binary is composed of 3 separate binaries; First Stage Bootloader (FSBL) elf, PL subsystem bitstream, and UBoot elf or user-built elf. These 3 binaries are integrated into a BOOT.BIN binary and it is built by the way descrbied in chapter 2.3.7.

At stage 2, First, FSBL initializes basic hardware built in Vivado block design, downloads FPGA bitstream to PL subsystem, and jumps to either UBoot or user-built elf application. If FSBL jumps to UBoot, UBoot will finally load Linux OS and jumps to the start point of Linux. FSBL also can load user-built standalone elf binary and jump to it. This standalone executable can be used to verify custom hardware in PL subsystem. We will replace the user built binary (or UBoot binary) with SLOS binary. We are going to build our operating system as an elf executable and embed it to the third partition of the BOOT.BIN. By following this way, we don’t need to take care of hardware initializations. Those are done in FSBL. Moreover, we can use a verified bootloader. Hardware initialization, bootloader are indispensable software in OS development, but they are non-operating system part and takes much time to implement if we do that from scratch. By tweaking the original booting sequence of Zynq7000, we don’t need to spend our time in non-operating system stuffs.

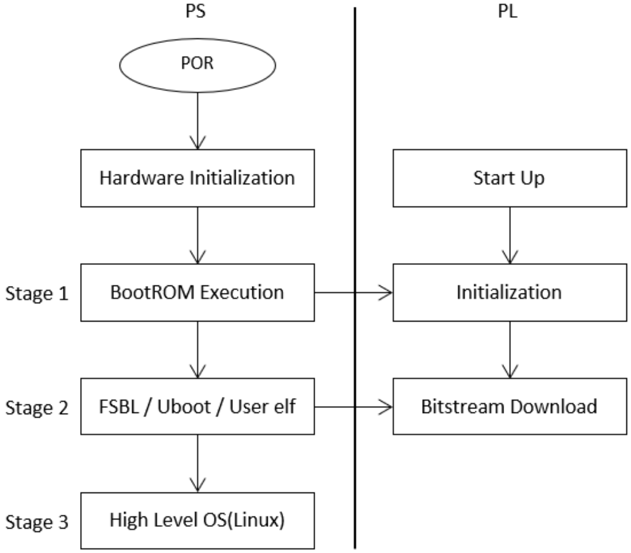


Figure 2-23: Zynq7000 booting sequence

2.7.3 Zynq7000 Evaluation Board

As mentioned, SLOS development uses the Zynq7000 evaluation board. The Zynq7000 evalution board looks like figure 2-24. The broad features are listed as follows [3].

1) Zynq XC7Z020-1CLG484C device
2) 1G DDR3 component memory (four 256 Mb x 8 devices)
3) Secure Digital (SD) connector

4) USB JTAG interface using a Digilent module
5) Clock sources
    a) Fixed 200MHz LVDS oscillator (differential)

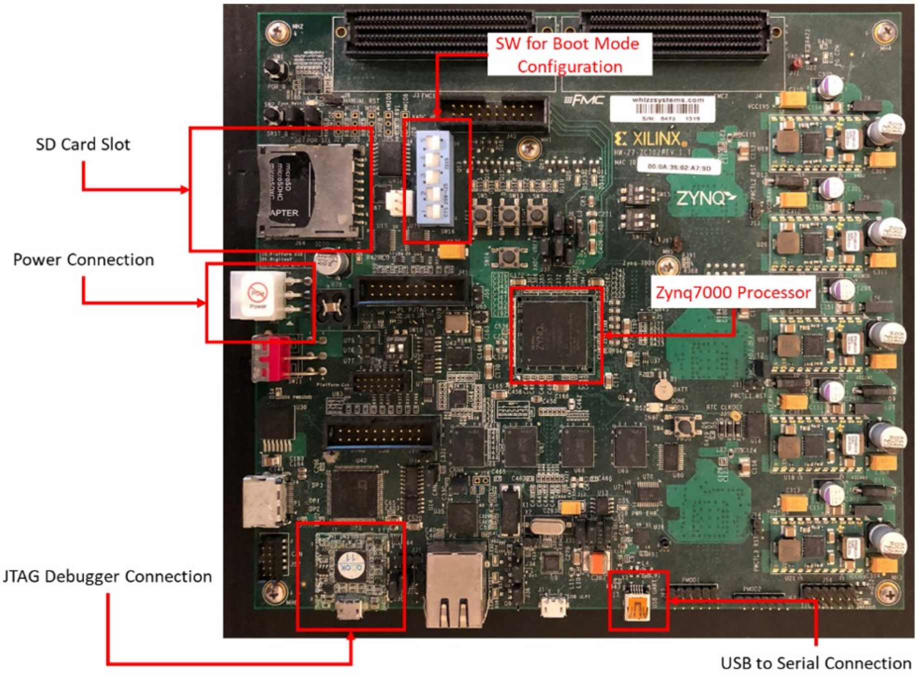


Figure 2-24: Zynq7000 evaluation board layout

    b) I2C programmable LVDS oscillator (differential)
    c) Fixed 33.33MHz LVCMOS oscillator (single-ended)
6) Ethernet PHY RGMII interface with RJ-45 connector
7) USB-to-UART bridge
8) HDMI codec
9) I2C bus
10) Status LEDs
11) User I/O
    a) Two programmable logic (PL) pushbuttons
    b) PL user DIP switch
    c) Eight PL user LEDs
    d) Two processing system (PS) pushbuttons
    e) Two PS LEDs
    f) Dual row Pmod GPIO header

   g) Single row Pmod GPIO header

12) AP SoC PS Reset Pushbuttons
    a) SRST_B PS reset button
    b) POR_B reset button
13) Two FMC LPC connectors
14) Power on/off slide switch
15) Dual 12-bit 1MSPS XADC analog-to-digital front end
16) Bootup configuration options

There are other commercial evaluation boards with Zynq7000 chipset. Those boards could also be used to follow up the steps in this book.

2.8 Summary

In this chapter, we set up the SLOS development environment. First, we set up the host computer by installing virtualbox with Ubuntu, tool chains and Petalinux. We also installed the Vivado IDE, SDK, Windows Git and serial terminal application into the host computer. We also created a default bitstream and built bare Linux image and booted the target board with them.

Operating system development needs some knowledge on hardware, especially on the CPU processor. SLOS runs on top of ARMv7 Cortex-A9 processor. We skimmed through the ARMv7 architecture, some essential part of ARMv7 ISA for future development and general-purpose registers and special purpose registers. Now that SLOS doesn't support the full features of ARMv7, for example, SLOS doesn't support the privilege mode switch between user mode and kernel mode.

The Xilinx Zynq7000 evaluation board is used for our SLOS development. It has a PS subsystem built with Cortex-A9 dual core integrated with other subsystems and Xilinx 7 series Artix-7 for PL subsystem. It has a well-defined debugging method such as Jtag and peripheral devices for serial UART. This evaluation board is booted from the BOOT.BIN in the SD card. We also defined how to build the BOOT.BIN image including the SLOS kernel elf.

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

[7] Petalinux and SDK Download:
https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools.html

[8] Vivado Download: https://www.xilinx.com/support/download.html

[9] GNU Cross Compiler:
https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads

[10] ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition,
http://infocenter.arm.com/

[11] Cortex-A9 Technical Reference Manual, http://infocenter.arm.com/

[12] Cortex-A9 MPCore Technical Reference Manual, http://infocenter.arm.com/

3 Building Sources and SLOS Boot Up in Zynq7000

3.1 Introduction

After chapter 2, we set up the development environment and get a shallow but essential background knowledge on the processor that we are going to work with. From chapter 3, we start to develop the SLOS. Since we finished the setup of development environment, next step is to bring up the board with a smallest set of SLOS and to print the famous "hello world" message to the 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. Basically, SLOS uses a GNU Make for its build system. The *Makefile* and *linker script* are added to this basic SLOS. This chapter will cover the basic part of GNU Makefile to understand the SLOS build system. The basic Makefile and build system will evolve as new features are added. Chapter 3 covers only basic "hello world" version of them.

Download the source from the github repository by running below command. Basic features of Git and core commands needed for SLOS development are described in chapter 1.5. The second command below checks out the chpater 3 sources of SLOS. *SLOS_CH3* is a branch name for this chpater. Your local branch is optional, it can be left out in the second command.

```
git clone https://github.com/chungae9ri/slos
git checkout -b your_branch_name SLOS_CH3
```

This source is only for your reference. It is the best for you to develop your custom operating system while we go through this book. You can also push your commits to fix a bug in SLOS into the github. Any kinds of contribution are appreciated.

3.2 Base SLOS Sources

Let's first build an initial source tree for our own operating system. If you follow the previous steps to pull the sources and checkout the *SLOS_CH3* branch, you should have the source tree as figure 3-1.

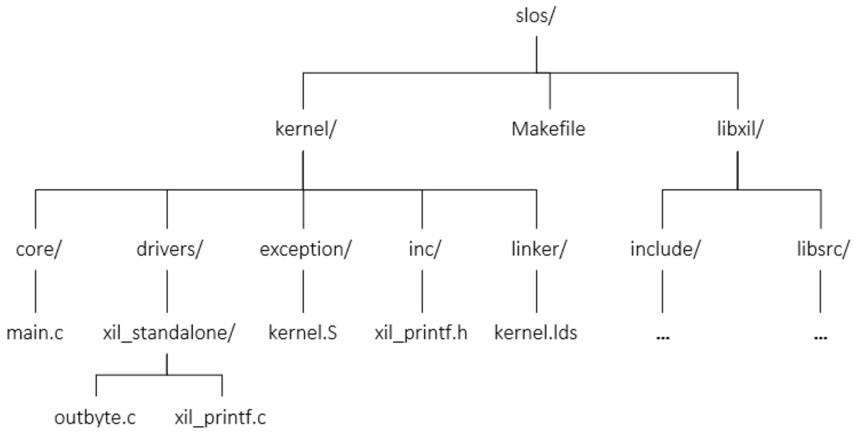


Figure 3-1 : SLOS base source tree for Helloworld

Kernel is a SLOS's top level directory which has other sub-directoires; *core, drivers, exception, inc* and *linker* directories. Kernel directory will contain files for core implementations of SLOS such as process management, memory management and so on. We are going to add more implmentations into these directories but this structure doesn't change through following chapters. The output binary of SLOS (kernel.elf) is created by building this directory linked with some static library files.

1) *kernel/core*
   Core directory is a source placement for kernel core features; process management, memory management, storage management, and other architecture dependent sources for implementing kernel core abilities. In chapter 3, the *core* directory contains only *main.c* and other kernel core files will be added to this directory in later chapters.

2) *kernel/drivers*
   Drivers directory has base BSP (Board Support Package) files such as Xilinx UART device driver, custome hardware device driver. Currently, this directory has only Xilinx UART device driver. This UART driver is used to print a message to the serial console. At the end of this chapter, we will print "hello world" message into the output screen.

3) *kernel/exceptions*
   This is a directory for exception vectors and fault handlers. Currently, only reset vector handler is all for exception vectors, which does nothing but jumping to the main entry point of SLOS kernel.

4) *kernel/inc*
   This is an *include* directory for header files. In SLOS_CH3 branch, header file for Xilinx UART device driver is placed in this directory.

5) *kernel/linker*
   This directory holds a linker script file.

There is a *libxil* directory in the same level as kernel directory. This directory is separated from kernel and supply the SLOS kernel with the basic BSP driver functionalities. Those drivers are needed for Xilinx specific hardware and are built as a library which is statically linked with the kernel object files. Xilinx already developed their standalone device drivers and we can get them from their SDK. If we need more Xilinx device driver, put that into this location and link it. All Xilinx device drivers will be compiled into a *libxil.a* library. SLOS uses only Xilinx UART device driver through the whole of this book. We will add a custom device driver in chapter 7 but it will be in *kernel/drivers* directory because it is custom device drivers for the custom peripheral hardwares that we will build.

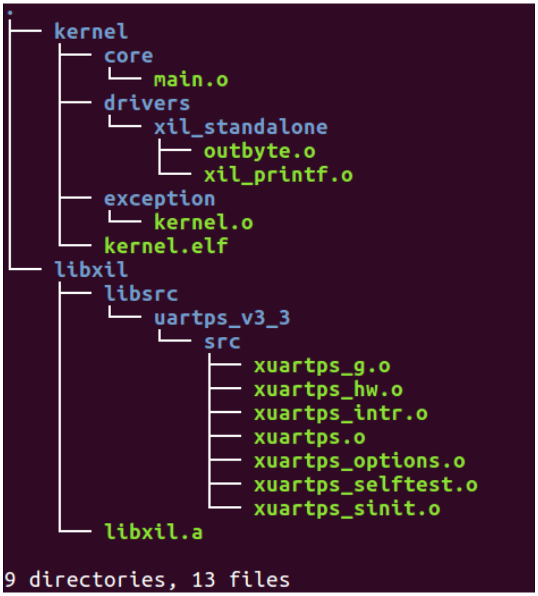
1) *libxil/libsrc/uartps_v3_3/src*
   This contains the source files for Xilinx UART device driver. These files are directly coming from Xilinx Zynq BSP. We don't have any interests in working on the development of the Xilinx UART device driver. Then, let's just borrow it from Xilinx.

Makefile defines how to build SLOS sources and how to create all output binary images. Makefile in chapter 3 builds libxil library first and moves to kernel directory and creates kernel.elf. Overview of Makefile is also covered in chapter 3.7.

In chapter3, these basic sources will bring up only Zynq7000 PS subsystem and print a simple message. There is nothing I need to do for PL subsystem until chapter 7. Hardware initialization is done in FSBL bootloader and we don't need to care about PS hardware initialization (e.g., System Level Control Register initialization). This is very important in custom operating system development because hardware initialization itself is not the essential area of operating system development, but very painful to implement, and must be solved 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 done just by reusing chipset vendor's bootloader and by correctly interfacing the bootloader with your operating system.

For your convenience, let's clone the SLOS Git repository into the host Windows' C drive or D drive. Then you can access it from your guest Ubuntu OS by simply *'cd ~/c/slos'* or *'cd ~/d/slos'* if you finish up the development environment setup. We build the SLOS source files from Ubuntu guest OS and debug it in host Windows OS using Xilinx SDK tools. Setting up this environment was covered in chapter 2.3.

If you correctly setup the developement PC, and pulled the SLOS sources, then, let's build it now. From your Ubuntu guest OS, run a command terminal (pressing Ctrl + Alt + t is a shortcut to launch terminal). Go to the SLOS source home directory. In my case, it is in Host Windows *C:\dotori\slos*. Run a *'make'* command and see if it succeeds in building the sources. If make finishes its job properly, there is a new directory named *'out'*. This out directory has the same directory tree like the source directory tree. The out directory has output object files, libxil.a and kernel.elf. Below figure 3-2 is the tree of out directory after the build is successfully done.



```
.
├── kernel
│   ├── core
│   │   └── main.o
│   ├── drivers
│   │   └── xil_standalone
│   │       ├── outbyte.o
│   │       └── xil_printf.o
│   ├── exception
│   │   └── kernel.o
│   └── kernel.elf
└── libxil
    ├── libsrc
    │   └── uartps_v3_3
    │       └── src
    │           ├── xuartps_g.o
    │           ├── xuartps_hw.o
    │           ├── xuartps_intr.o
    │           ├── xuartps.o
    │           ├── xuartps_options.o
    │           ├── xuartps_selftest.o
    │           └── xuartps_sinit.o
    └── libxil.a

9 directories, 13 files
```

Figure 3-2: Tree of output directory for Helloworld

As you can see, the out directory has exactly same tree structure as the SLOS source tree. Only the output files such as object files, executable binary, library file are placed in this output directory. Cleaning up the source tree can be done easily by deleting the out directory. Notice that the 'kernel.elf' file is the SLOS monolithic kernel binary.

3.3 Linker Script

Linker script is a recipe for a *linker* to create the final targe image by putting the object files, libraries together into one output file. Compiler compiles source codes and generate object files. Compiler puts source file's code, data to object file's specific *sections* (*TEXT*, *DATA* and so on). After compiling a source file, you can see the sections in an object file by using the *arm-none-eabi-objdump* program with the *'-h'* option in the Ubuntu guest OS. Then, linker links these object files by solving function and variable references and puts the input sections into a proper output sections to generate a final executable. While doing this, linker's main roles are

1) Merging input sections from input files to output sections
2) Mapping memory layout of output sections

These two roles are all that linker does. Linker script defines the upper 1) and 2) for the linker. Linker can do more but in SLOS development, we need upper two things only; input-output section mapping and memory layout control. After completing the first linker script for "hello world", we are going to edit this linker script as new features are added to SLOS. Let's have a quick look at how linker script syntax looks like and how it can be used in the SLOS development. Most of the below sections comes from reference [1] and [2].

3.3.1 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 called 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 not loadable nor allocatable typically contains some sort of debugging information.

Sections needed to run a program is also called *segment*. This information is necessary for a program launcher to execute a runnable program. There is a *PHDRS (Program Headers)* command to define custom segments for the linker. But if customized segment is not defined, linker uses its default predefined program header for segments. SLOS also doesn't define its program headers.

Every loadable or allocatable output section has two addresses; *VMA (Virtual Memory Address)* and *LMA (Load Memory Address).* The VMA is the address that the section will have when the output file is run. The LMA is the address at which the section will be loaded. In most cases the two addresses will be the same. Until chapter 4, we use the same VMA and LMA. But after chapter 5 memory management, SLOS will use different LMA and VMA to load kernel to the virtual address at 0xC000_0000. The Helloworld version of SLOS in this chapter has the same VMA and LMA.

Every object file also can have 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 and other information. If you compile a C or C++ program into an object file, you will get a defined symbol list for every function, global variables and static variables. 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 script is a just text file. A linker script is written 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.

3.3.2 Helloworld Linker Script

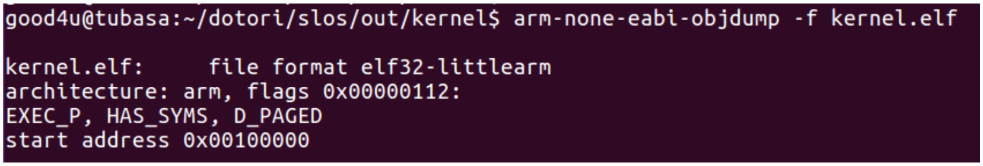
Below is the simplest linker script for Helloworld SLOS running in Zynq7000 evaluation kit. This simple linker script is good enough to run our Helloworld SLOS operating system. Let's call this base linker script as "Helloworld" linker script. We don't need to change much from below base linker script and the basic structure of this linker script remains through this book. Big changes will happen in chapter 4 memory management to make the VMA and LMA be different. The Helloworld linker script is located in *kernel/linker/* and named as *kernel.lds*. Let's have a look into the things of what we need to know about linker script for our SLOS development.

```
OUTPUT_ARCH(arm)
ENTRY(exceptions)

MEMORY
{
    ps7_ddr_0_S_AXI_BASEADDR : ORIGIN = 0x100000, LENGTH = 0x3FF00000
}

SECTIONS
{
    . = 0x100000;
    .text : {
                    *(EXCEPTIONS);
                    *(.text)
    } > ps7_ddr_0_S_AXI_BASEADDR
    .data : {
                    *(.data);
```

```
	} > ps7_ddr_0_S_AXI_BASEADDR
	.bss : {
			*(.bss);
	} > ps7_ddr_0_S_AXI_BASEADDR
}
```

1) OUTPUT_ARCH(arm)

This command specifies a particular output machine architecture. The argument is one of the names used by the *BFD (Binary File Descriptor)* library [3]. You can see the architecture of an object file by using the *arm-none-eabi-objdump* program with the *'-f'* option. The architecture name *"arm"* in below screen capture shows the output architecture.



```
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
```

Figure 3-3: Output architecture of SLOS

2) ENTRY(exceptions)

ENTRY command is used for setting the entry point of a program execution. Entry point has the first instruction of the program to be run. The argument *exceptions* is a lable of an address which is defined in *kernel.S*. SLOS defines exceptions a label as the start address of exception vectors which is located at 0x0010_0000.

There are several ways described in below 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.

a) The *'-e'* command-line option for the linker. This option is used when linker executable is running.
b) The *ENTRY(symbol)* command in a linker script as in Helloworld linker script.
c) The value of the symbol *start*, if defined.
d) The address of the first byte of the *.text* section, if present.
e) The address 0.

3) 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. It consists of region name, start address and length. MEMORY command format 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. In the Helloworld linker script, the *ps7_ddr_0_S_AXI_BASEADDR* in line 6 is a memory region name.

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

a) *R*: Read-only section
b) *W*: Read/write section
c) *X*: Executable section
d) *A*: Allocatable section
e) *I*: Initialized section
f) *L*: Same as I
g) *!*: 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 SLOS linker script, there is only one memory region which is defined as DDR memory region and its *ORIGIN* and *LENGTH* comes from the system-level address map in Zynq TRM document. MEMORY command is optional, and it can be left out from the Helloworld SLOS linker script.

4) SECTIONS Command

SECTIONS command is the most important command in linker script. SECTIONS command does the principal things that the linker script must do. First, it maps input sections to output sections. Second, it places 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 Helloworld linker script sets the value of the special symbol *'.'*, which is called as *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 11 in the example linker script sets current location counter value as 0x0010_0000. After that, the location counter is increased by the amount of output section size or you can set it with a specific absolute address value.

As mentioned earlier, every loadable or allocatable output section has below two addresses.

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

Up to chapter 4 in this book, VMA and LMA addresses remain same. But after chapter 5, memory management, the VMA and LMA address gets different. Helloworld SLOS has the same VMA and LMA.

.text in line 12 designates the text output section. Following lines in text section build the .text output section which is composed of EXCEPTIONS input section and .text input sections from all input files. The wild card *'*'* is used for any file names. That is, *(.text) in line 14 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 0x0010_0000. This SECTIONS command does the mapping from input sections to output sections and compose the memory layout of output sections.

.data section and *.bss* section are handled in the same way as .text section. All input data sections of input files go to data output section and all bss input sections for uninitialized data go to bss output section.

After you successfully build the Helloworld SLOS kernel source in chapter 3.1, you can check 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 [2] for more details on linker script, but the basic Helloworld SLOS linker script is enough for this simple operating system development. At the end of chapter 3, we can run Helloworld SLOS in the Zynq7000 target board with this simple linker script.

3.4 Exception Vector and Reset Handler

Processor has an exception vector for handling asynchronous events from the processor. '*Asynchronous*' means this event happens independently as the program sequence flows. One example of those asynchronous events is an *interrupt*. If such event occurs, the processor stops the execution of current program 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. Refer chapter 2.5 for the detailed description about these exceptions.

In this chapter, we are going to implement a simple exception vectors which has only a reset vector handler. Since the handlers in exception vector have only 4 bytes length, all they can do is jumping to the detailed implementation routine for handling each exception. Below Table 3-1 summarizes the exception vector used for the SLOS operating system development.

Exception	Offset	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	

Table 3-1: List of exception vectors

Normally, the reset handler code in ARM must do some, or all of the following:

1) In a multi-core system, enable non-primary cores to sleep.
2) Initialize exception vectors.
3) Initialize the memory system, including the MMU.
4) Initialize core mode stacks and registers.
5) Initialize any critical I/O devices.
6) Perform any necessary initialization of *NEON* or *VFP*.
7) Enable interrupts.
8) Change core mode or state.
9) Handle any set-up required for the Secure world.
10) Call the *main()* application.

When there are muliple cores in the processor, a common approach of the SMP OS is that reset is to permit a single core within the cluster to perform system initialization, while the same code, if run on a different core, will cause it to sleep, that is, enter *WFI (Wait For Interrupt)* state. The other cores might be woken after core 0 has created a simple set of memory management, as these could be used by all cores in the system. Then, the secondary cores are typically woken up later by an SMP OS.

In this chapter, we will do only the 10), call the main() function, in the upper list. Even though SLOS doesn't do anything on the secondary core, but it boots up very well. Although the reset handler in Helloworld SLOS doesn't do anything but jumping to main, we need to know more on the initialization of exception vector, 2) in the upper list, because we are going to add new exception handlers such as interrupt handler from chapter 4.

The start address of exception handlers can be set by *VBAR (Vector Base Address Register)* register and each handler placed at 4 bytes offset from this base address. The VBAR is set by *System Control Register, cp15.* Since our Helloworld linker script sets the exception address at 0x0010_0000, and put the exception handler code into that address, we have to set the VBAR value with 0x0010_0000. The VBAR can be set as below code.

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

The second line uses a special ARM instruction *(mcr)* to set the system level control register. You can refer ARM TRM document [4] to learn these system level registers (ARM coprocessor registers).

The Helloworld operating system implements only reset vector and all other exception handlers are left blank for now. Our reset vector handler doesn't do anything meaningful operations, it only jumps to main function. Now, it's time to do some ARM assembler programming for the first time. Let's add kernel.S file to the *kernel/exception* directory and add below base code for reset vector to kernel.S. If you already pulled out the SLOS sources from github, you can just refer the *kernel/exception/kernel.S* file in your SLOS Git repository.

```
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
```

Can you see how simple it is? Line 1 is for comment which is same format as C programming language. Line 2 is for specifying the extern main() function symbol. In this case, the symbol of kernel main() function comes from other file and it is used for the jump address at the end of the reset vector. Line 3 is for specifying the section name of the routine. Linker script will put this section of input file kernel.S into the first place of output section which must be 0x0010_0000. Line 4 is for specifying ARM architecture and Line 5 is for specifying the global symbol (or address) of the exception vector. This symbol is used for the linker script (line 2) to set the entry point of Helloworld SLOS.

At this moment, the entry routine of exception vector which is a reset handler needs to be implemented. As mentioned earlier, the reset vector just jumps to other address (line 7) which is the beginning of detailed reset handler implementation. The *reset_handler* symbol is the address of real beginning of reset vector handler implementation. The Helloworld SLOS has the reset vector handler only jumping to the kernel main entry address that is imported in line 2. This reset handler is very simple. It does nothing but jumps to the kernel main entry in the list of reset handler works mentioned before.

3.5 Kernel Main Entry Point

Kernel main entry point of the Helloworld SLOS prints a “hello world” message to the console screen. The console screen is set in chapter 2.3.3. This console screen is used standard I/O in SLOS. Then, the main() function falls into infinite idle loop. It only prints a simple message and then spins forever. Kernel main entry implmentation looks like below.

```
#include <xil_printf.h>

void cpuidle()
{
/* do nothing for now */
    for (;;);
}

int main(void)
{
    xil_printf("###hello world\n");
    cpuidle();

    return 0;
}
```

Line 1 includes xil_printf.h file for using Xilinx UART driver. This UART driver sends a character string to user console terminal connected through USB-to-Serial connection. How to include the UART driver sources will be explained in next chapter. Line 3 to line 7 is the cpuidle() function which has the ARM processor spin forever. The kernel 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(). This main loop never returns.

This main() looks very simple, but it includes many pre-requisites to make it work. First, we need to define development environment as in chapter 2. It is development board, cross compiler toolchain, serial terminal and so on. Second, we need to know what the linker script is and how to make it to place the compiled code into memory region. Third, we should have basic idea on the ARM processor architecture, at least the reset vector handler. Fourth, we don't cover yet but we need to know how to implement the UART device drvier and finally we need to figure out how to build the source files. Printing "hello world" doesn't sound that easy now. The fourth and final work will be covered in next section.

3.6 Porting Xilinx UART Drivers

As described in chapter 1, operating system must drive system hardware resources. As adding more operating system features into this simple Helloworld SLOS, we are going to bring up essential hardwares such as *GIC (generic interrupt controller),* Timer, MMU and so on. In the Helloworld SLOS development, we need a device driver for UART to send message to serial terminal in the host PC. This device driver is a minimum requirement, but this driver needs hardware specific information and developing a Xilinx device driver is not our interest. We will practice hardware-software codesign later with our custom hardware design. We will design our own hardware and add its device driver into SLOS. Since the only thing that Helloworld SLOS does is printing "hello world" message, it needs only UART driver for now. The sources for Helloworld SLOS downloaded from the Git repository has already UART device driver in *libxil/libsrc* and *kernel/drivers* folder. So, if you successfully downloaded the base sources, you can use it without any concern of Xilinx UART implementation.

Normally, we don't need to know how Zynq7000 peripheral hardware works. Nonetheless, it is good to know how to port those pre-developed, verified sources to our custom SLOS project. Xilinx already developed standalone device driver sources for their peripheral hardwares. Let's first port a verified Xilinx UART device driver to our BSP. The first step for this is to make their basic BSP libraries. After you correctly generate the default block design as in chpater 2.3.7, then you can generate a standalone BSP. First, generate the FPGA bitstream for basic Zynq7000 hardware as described in chapter 2.3.7, and follow below steps to generate the Xilinx standalone BSP sources.

1) In the default vivado project in chapter 2.3.7, go to File -> Launch SDK. Then just

click OK to choose default settings in the popup window as in figure 3-4.

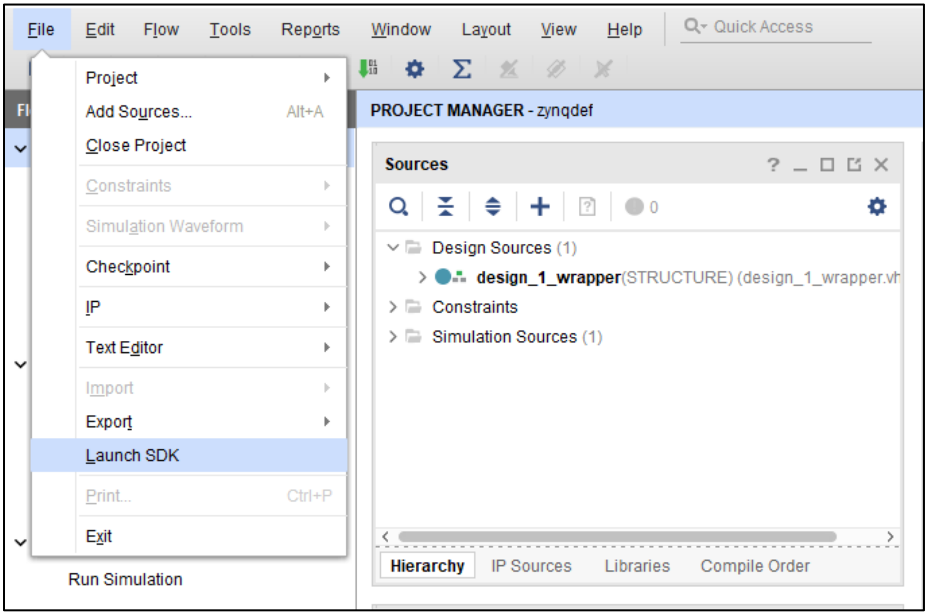


Figure 3-4: Launch Xilinx SDK

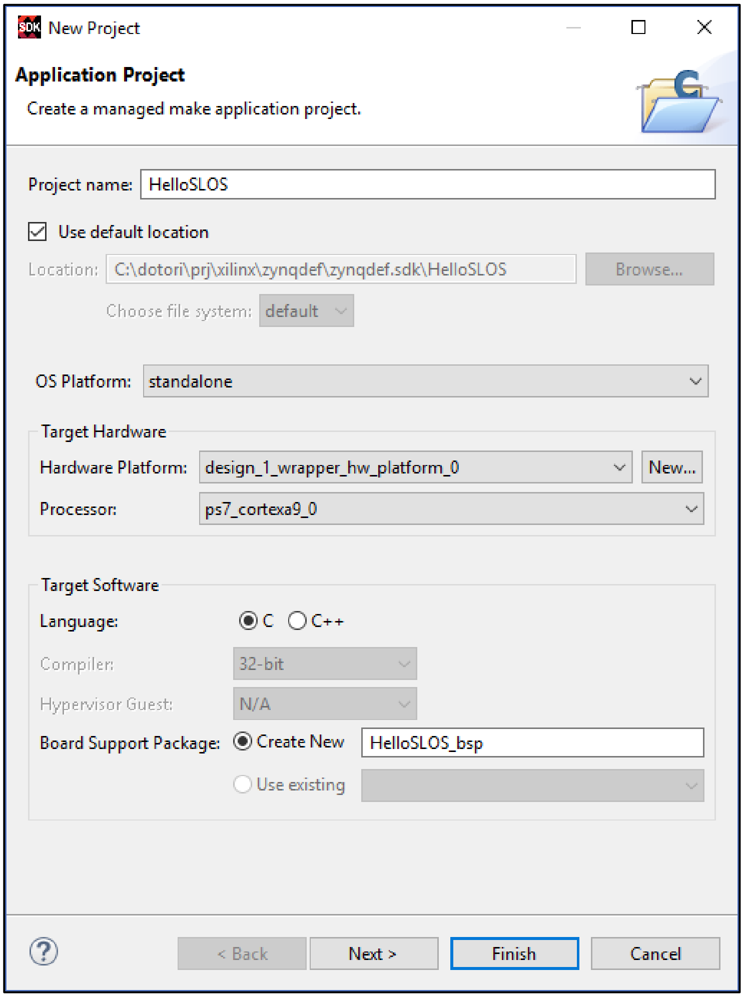
2) The step 1 will launch an *Eclipse* IDE. The Eclipse already has a PS system initialization source files and register initialization routines. The Zynq FSBL bootloader runs the initialzation routines in this project.

3) Now, let's create a standalone BSP sources. In the Eclipse IDE, go to File -> New -> Application Project. Then, input the project name as 'HelloSLOS' and make all other settings left by default. It looks like figure 3-5. Click Finish button.

4) Step 3 creates two new projects which are *HelloSLOS* and *HelloSLOS_bsp*. The *HelloSLOS* project has a main application sources that prints "hello world" message. The *HelloSLOS_bsp* is a standalone BSP project that covers the Zynq7000 hardware. All standalone device drivers in *HelloSLOS_bsp* are built into *libxil.a* library. We are going to leverage this *HelloSLOS_bsp* project to port the UART device driver to our Helloworld SLOS.

5) Go to *HelloSLOS_bsp/ps7_cortexa9_0/libsrc* and copy the *uartps_v3_6* folder to our *libxil/libsrc*. That's it! Driver version number could be different based on the Xilinx SDK installation version.

If you want other device drivers, then copy them also from *HelloSLOS_bsp* to the same path. But we only need a UART driver for printing "hello world" message from our own SLOS. We don't use the rest Xilinx device driver through the end of this book.



Application Project

Create a managed make application project.

Project name: HelloSLOS

☑ Use default location

Location: C:\dotori\prj\xilinx\zynqdef\zynqdef.sdk\HelloSLOS

Choose file system: default

OS Platform: standalone

Target Hardware

Hardware Platform: design_1_wrapper_hw_platform_0

Processor: ps7_cortexa9_0

Target Software

Language: ● C ○ C++

Compiler: 32-bit

Hypervisor Guest: N/A

Board Support Package: ● Create New HelloSLOS_bsp

○ Use existing

Figure 3-5: Bare Application Project settings

3.7 Makefile and SLOS Build Process

3.7.1 Overview of Makefile

GNU Makefile is a collection of recipes to guide the GNU make to perform the orders in the Makefile. It defines how GNU make compiles sources, how it links the object files and finally how to generate the final executable binary. It could be used to run other work orders but widely used for build system. Knowing GNU Makefile itself takes time, but we are going to use a small portion of the features necessary for our development. Let' have a look at those essential parts.

First step is to get basic idea on how GNU make 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. In the second phase, GNU make uses these internal structures to determine what targets will need to be rebuilt and to invoke the rules necessary to do so [5]. 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 occurs in second phase, it is *deferred assignment*. We use *':='* operator for immediate assignment and *'='* operator for deferred assignment.

The basic Makefile needs *rules* that define how to generate the output of Makefile. It looks like as below:

```
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 name.

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 GNU make to rebuild the target.

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

Below is an example Makefile to build hello target executable.

```
all : hello
hello : main.o hello.o
                gcc -o $@ $^ -lc
main.o : main.c
                gcc -c $^
hello.o : hello.c
```

```
7                       gcc -c $<
8      clean :
9                       rm *.o hello
```

The GNU make first search the first target which is *all* target in line 1, then it will find the *all* target's dependency on *hello*. Now GNU make creates the dependency graph for all these targets through line 2 to line 7. This is the phase 1 work of GNU make. Then, make will generate those targets with bottom-up traversing the graph and apply the recipes to build targets. This is the phase 2 work of GNU make. In this example, building *hello* depends on *main.o* and *hello.o* which are depends on again *main.c* and *hello.c.* So, GNU make first tries to build *main.o* and *hello.o* by using the recipes defined in line 5 and line 7. Then, traverses back to the dependency tree up to the *hello* target.

The value of variable is referenced by using *'$'* symbol in the Makefile. But in this simple Makefile, the *$* symbol is used for other purpose of *Automatic Variables.* The reference document [5] explains more about these automatic variables. Below three are the automatic variables used in the example Makefile.

1) *$@*
The file name of target of the rule.

2) *$^*
The names of all prerequisite files with white space between them.

3) *$<*
The name of first prerequisites.

The line 3, *-lc,* is a linker option used for linking a library whose file name is *libc.a.* Be careful about the library name in the linker option which uses only after '*lib'* and the file extension is left out. For example, if the library name is *libhello.a*, then the linker option for linking this library must be *-lhello*. Linker option *-L* defines the library search path. We are going to use these few linker options but there are many other options you can find in the document [2].

3.7.2 SLOS build process

Figure 3-6 is the build process of Helloworld SLOS. This build process looks very simple but used until chapter 6.7. We will add a ramdisk binary and change this build process in chapter 6.7.1. Until then, we stick to the build process in figure 3-6 and we only add more source files and generate more object files for different SLOS features.

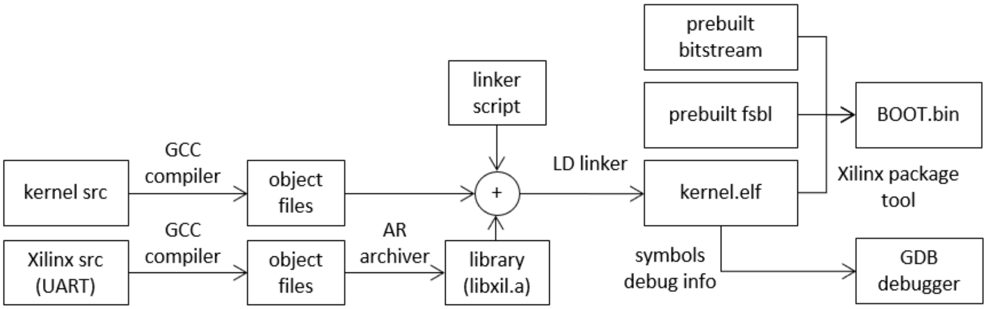


Figure 3-6: Helloworld SLOS build process

The Helloworld SLOS has only one kernel source file: main.c. The build process for other kernel subsystems such as process management, memory management stays in the same place as in figure 3-6, because those are implemented just by adding new source files into the Helloworld SLOS's kernel directory.

As in figure 3-6, the Makefile of Helloworld SLOS describes the steps for building libxil.a, kernel.elf. After Makefile generates the kernel.elf and libxil.a*,* Xilinx SDK tool *(petalinux-package)* mentioned in chapter 2.3.7 merges prebuilt PL bitstream and prebuilt FSBL binary with SLOS kernel.elf to generate final BOOT.bin image. Below Makefile is the steps to generate the SLOS's kernel.elf. GNU make finds the source files and builds target by using the recipes defined in the Makefile. So, we don't need to change this Makefile even though new source files regarding new SLOS features are added. Since this Makefile is a basic structure used through whole of this book, let's look into it more deatil.

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

LIBS := $(HOME)/bin/arm-2017q1/arm-none-eabi/lib
LIBS2 :=$(HOME)/bin/arm-2017q1/lib/gcc/arm-none-eabi/6.3.1
CC := arm-none-eabi-gcc
ASM := arm-none-eabi-as
LD := arm-none-eabi-ld
AR := arm-none-eabi-ar
OBJCOPY :=arm-none-eabi-objcopy
LIBXIL := libxil.a

TOP_DIR :=$(shell pwd)
OUT_TOP := $(TOP_DIR)/out

KERNMODULES := core exception drivers/xil_standalone
KERNSRCDIR := $(addprefix kernel/,$(KERNMODULES))
KERNOUTDIR := $(addprefix out/kernel/,$(KERNMODULES))
```

```

KERNCSRC := $(foreach sdir,$(KERNSRCDIR),$(wildcard $(sdir)/*.c))
KERNCOBJ := $(patsubst %.c,out/%.o,$(KERNCSRC))
KERNASMSRC := $(foreach sdir,$(KERNSRCDIR),$(wildcard $(sdir)/*.S))
KERNASMOBJ := $(patsubst %.S,out/%.o,$(KERNASMSRC))

LIBMODULESTEMP:= uartps_v3_3
LIBMODULES:= $(addsuffix /src, $(LIBMODULESTEMP))
LIBSRCDIR := $(addprefix libxil/libsrc/,$(LIBMODULES))
LIBOUTDIR := $(addprefix out/libxil/libsrc/,$(LIBMODULES))
LIBCSRC := $(foreach sdir,$(LIBSRCDIR),$(wildcard $(sdir)/*.c))
LIBCOBJ := $(patsubst %.c,out/%.o,$(LIBCSRC))
LIBASMSRC := $(foreach sdir,$(LIBSRCDIR),$(wildcard $(sdir)/*.S))
LIBASMOBJ := $(patsubst %.S,out/%.o,$(LIBASMSRC))

INC := -I$(TOP_DIR)/kernel/inc -I$(TOP_DIR)/libxil/include
LDS :=$(TOP_DIR)/kernel/linker/kernel.lds

vpath %.c $(KERNSRCDIR)
vpath %.S $(KERNSRCDIR)
vpath %.c $(LIBSRCDIR)
vpath %.S $(LIBSRCDIR)

define make-obj
$1/%.o: %.c
	$(CC) $(CFLAGS) $(INC) -o $$@ -c $$< -g -mcpu=cortex-a9
	-mfpu=vfpv3 -mfloat-abi=softfp

$1/%.o: %.S
	$(CC) $(INC) -o $$@ -c $$< -mcpu=cortex-a9 -mfpu=vfpv3
	-mfloat-abi=softfp
endef

$(foreach bdir, $(KERNOUTDIR),$(eval $(call make-obj,$(bdir))))
$(foreach bdir, $(LIBOUTDIR),$(eval $(call make-obj,$(bdir))))

all: checkdirs $(LIBXIL) kernel.elf

checkdirs : $(LIBOUTDIR) $(KERNOUTDIR)

$(LIBOUTDIR) :
	mkdir -p $@

```

```
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)
                    $(KERNASMOBJ) -L$(OUT_TOP)/libxil -L$(LIBS) -L$(LIBS2) -lxil -lc -lgcc
67
68      clean :
69                  rm -rf $(OUT_TOP) libxil.a
70                  rm -f libxil/*.a $(LIBCOBJ) $(LIBASMOBJ)
```

Line 1 sets the path to the GNU tool chain. This path also can be added to the *.bashrc* file as described in chapter 2.3.4. Notice that we should use the immediate assignment to set up this toolchain path. If we use a deferred assignment (= symbol) for *$PATH* environment variable instead of using the immediate assignment (:= symbol), the GNU make will fail to find the GNU tool chain binaries in performing the recipes in its phase 2 and can’t build the target. So, it is important to understand the difference coming from the two phases of GNU make process.

Line 3~10 are for setting the GNU tool binaries, and the path to the libraries linked to kernel.elf. The line 65~66 links libxil.a*,* libc, libgcc together with bare kernel object files to generate kernel.elf. These libraries are searched in this library path.

Line 15~17 defines the subdirectories of kernel source files and the output folder of compiled object files. Any files with file extension *.c, .S* are to be automatically included into the source file list. We don’t need to add each file name into Makefile when adding a new source file into the kernel source tree. GNU make adds new files in these directories.

Line 19~31 lists up the source files which has .c or .S file extension from the directory defined in line 15~17. Line 19~22 generates a list for SLOS bare sources, and line 24~31 does the same job for Xilinx BSP library. In this simple startup phase, Xilinx BSP library has only UART device driver.

Line 33 is for defining the include path which is used in compiling the sources in line 41 and 47.

Line 36~51 are the rules to build the object file from source files in all directories. The *vpath* directive allows you to specify a search path for a particular class of file names: those match a particular pattern. This is used as a form of *vpath pattern directories* in line 36~39.

Line 52 is the target of this Makefile, which means this target is the top most location of the dependency tree. This target depends on the prerequistes; *checkdirs, $(LIBXIL)* and *kernel.elf*. The first prerequisite checkdirs defines how to create the output directories for kernel.elf and libxil.a under the out folder.

Line 62~66 defines the recipe that GNU make does for generating the Xilinx BSP library and kernel.elf. The libxil.a is generated by using GNU archiver *(arm-none-eabi-ar)* in line 63. The archiver merges the Xilinx BSP output object files to libxil.a file. Finally, the last dependency of target, kernel.elf, is generated in line 66. The GNU linker *(ld)* put all object files, libraries together in line 66.

Line 68 is another target of this Makefile which cleans all outcomes by removing all the output objects. These outputs are all located in the out directory.

This Makefile describes the whole process for creating kernel.elf and covers all path from kernel src, Xilinx src to build the kernel.elf in figure 3-6.

3.7.3 Funcation Call in a Makefile

Line 49 and 50 in the Helloworld SLOS Makefile use a function call. These functions allow you to do text processing in the Makefile to compute the files to operate on or the commands to use in the recipes. A function call in Makefile is similar with variable reference; the result of the function's processing is substituted into the Makefile at the point of the call. The syntax of function call looks like this:

$(call *function,argument1,argument2,...*)

function is a function name. When make expands the function, it assigns each argument to temporary variables $(1), $(2)... and so on. The variable $(0) will be the function itself. The functions used in this Makefile is explained below. For detailed information, and for other functions supported in the Makefile, refer document [5].

1) $(*addprefix prefix, names ...*)
   This function adds prefix to the series of names. Names is a series of string separated by white space. For example, the output of line 17 *$(addprefix kernel/,$(KERNMODULES))* in the *Makefile* is *kernel/core kernel/exception kernel/drivers/xil_standalone*.

2) $(*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.

3) $(*foreach var,list,text*)
   This function is used for repetitive iteration of *text* with *var* which is assigned values from the *list*. Line 19 in the Helloworld Makefile, *$(foreach sdir,$(KERNSRCDIR),$(wildcard $(sdir)/*.c))* means all *.c* files in each *$(KERNSRCDIR).*

Then, $(KERNSRCDIR) variable is assigned as *kernel/core kernel/exception kernel/drivers/xil_standalone* in line 17.

4) $(*patsubst pattern,replacement,text*)
   This function is for pattern substitution. It replaces the *pattern* in *text* with *replacement*. Line 20, *$(patsubst %.c,out/%.o,$(KERNCSRC))* replaces the *.c* extension of kernel source files with *out/src_file_name.o. '%'* is a wildcard character for matching any number of any characters within word.

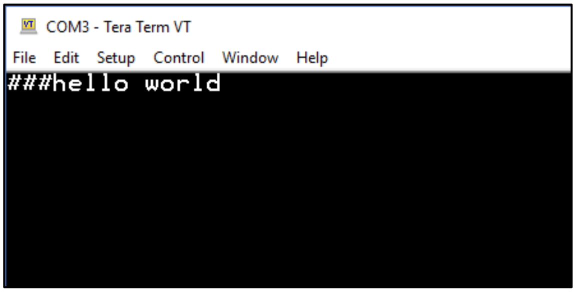
5) $(*eval ...*)
   eval function allows you to define new makefile constructs that are not constant; which are the result of evaluating other variables and functions. The argument to the eval function is expanded, then the result of that function is parsed as Makefile syntax. The expanded results can define new variables, targets, implicit or explicit rules.

6) $(*call variable,param1,param2,...*)
   This function can call user defined function with parameter *param1*, *param2*. param1 is referenced in the function as *$(1)* and param2 is referenced as *$(2)* and so on. The *$(0)* is for *variable* (user defined function name). 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.

3.7.4 Booting Helloworld SLOS

Now, it's time to boot up the Zynq7000 evaluation board with our simple Helloworld SLOS. We have covered below modules so far only to print "hello world" message to the screen.

1) Install Virtualbox, Ubuntu 16.04, Xilinx Vivado, SDK, Petalinux and set up development environment.
2) Build basic Zynq7000 block design and generate PL subsystem bitstream, zynq_fsbl bootloader.
3) Install GCC cross platform tool chain.
4) Develop a linker script for mapping from input sections to output sections.
5) Develop a makefile for build source files.
6) Port Xilinx UART standalone device driver.
7) Develop a reset handler and kernel main entry.
8) Build kernel.elf for Helloworld SLOS and package all output binaries with petalinux-package tool.



```
###hello world
```

Figure 3-7: “hello world” message from the simplest SLOS

After all these steps are done, you will have a BOOT.BIN file containing the simplest form of SLOS. Copy BOOT.BIN into the SD card and insert that SD card to Zynq7000 evaluation board. Turn on the power of the board. Following “hello world” message must be shown in the screen. That’s all we need until this chapter.

Now, we are ready to dig into the real part of operating system development. But there is one more important thing to do before embarking on our operating system developement journey. We need to set up a well-defined debugger. Having a good debugger is always important in software development. Let’s set up a debugger before we get stuck by any defects and get lost in the dark.

3.8 Debugger in SLOS

3.8.1 How to debug your operating system?

Well-defined debugger is always important in development. We have to prepare a good debugger before going further. Xilinx already developed a debugger named *XSDB (Xilinx System Debugger).* The XSDB debugger is installed in the 2019.2 SDK tool set. To use XSDB debugger, install the Vivado SDK as described in chapter 2.3.5 [6]. We also need to use the *GNU Debugger (GDB).* We are going to combine the XSDB debugger with the GNU debugger to debug the SLOS. XSDB is connected to ARM Cortex processor by using JTAG interface and export GDB interface to the remote debugger. Figure 3-8 shows the connections between GDB, XSDB and ARM Cortex-A9 processor in Zynq7000 PS subsystem.

XSDB and GDB are all included into SDK installation. If you properly install Vivado and its SDK in the previous chapter, you can use all SDK tools after running the *settings64.bat* file. The settings64.bat sets the environment to run SDK tools. Open a command window in your host Windows OS and go to your SDK installation directory (*C:\Xilinx\Vitis\2019.2* for example). You can see the settings64.bat file, then run it.

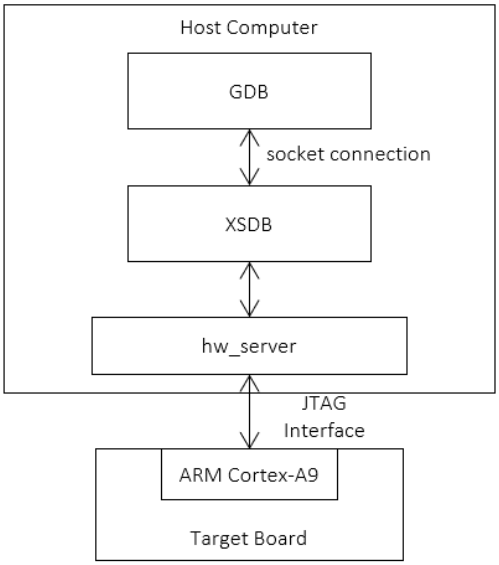


Figure 3-8: XSDB debugger connections

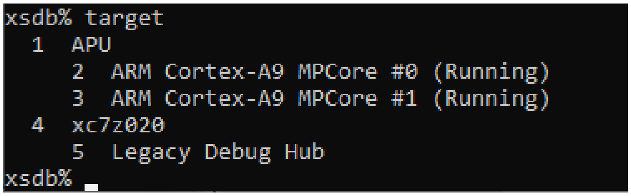
Now, let’s connect the debuggers to target board and break the processor at the reset vector which is the right start point of SLOS. Turn on the power in the target board, then you can see the “hello world” message shown in the terminal screen. At this point the cpu is spinning forever in the loop of cpuidle() function. Before going further, you need to check the physical cable connection between host computer and target board. There should be two USB cable connections. Those are UART-to-Serial connection and JTAG connection. After connecting those two USB connections between your host PC and the target board , follow below steps to connect debugger.

1) Run Windows command console. Go to Xilinx installation directory (in this case C:\Xilinx\Vitis\2019.2) and run ‘settings64.bat’ file. This batch file sets up the console environment for debugger.

2) Then run ‘*hw_server’* command. Normally, when the Xilinx USB cable is connected, the hw_server starts automatically. If it doesn’t, run it manually. The hw_server bridges the communication between desktop debugger to target board through JTAG. Now, the Xilinx JTAG debugger (Platform Cable USB) should show green light. If this still amber color or red color, run Vivado 2019.2 and connect target in the ‘Opne Hardware Manager’. This should make it green.

3) In the console, run *'XSDB'* command to launch a Xilinx System Debugger. In the command prompt, run '*connect*' to connect to *hw_server/TCF agent* which was set in step 2).

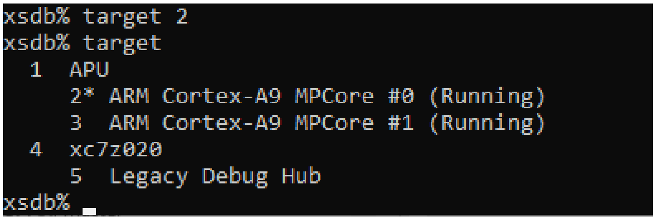
4) Run *'target'* command to list up the possible targets. You are supposed to see messages like following figure 3-9.



```
xsdb% target
  1  APU
     2  ARM Cortex-A9 MPCore #0 (Running)
     3  ARM Cortex-A9 MPCore #1 (Running)
  4  xc7z020
     5  Legacy Debug Hub
xsdb% _
```

Figure 3-9: XSDB list up daisy-chained target processors

5) Connect a target processor by running '*target ${Num}*'. ${Num} is a number variable listed in step 5). In our case this is 2, ARM Cortex-A9 MPCore #0. If target processor is properly connected, there is a '*' in the list as in figure 3-10. After this, we can run Xilinx debug commands. In chapter 8, we will connect the JTAG to MPCore #1 for debugging SMP SLOS.



```
xsdb% target 2
xsdb% target
  1  APU
     2* ARM Cortex-A9 MPCore #0 (Running)
     3  ARM Cortex-A9 MPCore #1 (Running)
  4  xc7z020
     5  Legacy Debug Hub
xsdb% _
```

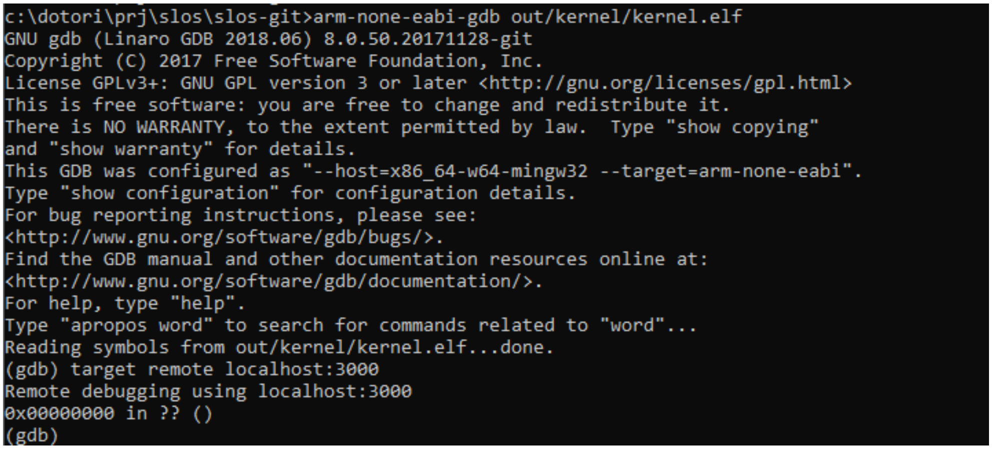
Figure 3-10: Target processor is selected for JTAG debugging

6) Let's break the processor at the address 0x0 by running *'rst'* command. Let's add the breakpoint at 0x0 and 0x10000. 0x10000 is the reset exception vector address. Run 'bpadd 0x0' and 'bpadd 0x100000' and run 'rst' command in the XSDB window. The core #0 should be reset and stopped at 0x0. 'bpadd' and 'rst' command will be covered in next chapter.

7) Now, let's connect the GDB to the XSDB. XSDB will serve as a gdb server. Launch another Windows command console, go to Vitis directory and run 'settings64.bat'.

8) Go to the SLOS's build directory and run *'arm-none-eabi-gdb out/kernel/kernel.elf'*. Or you can launch *kernel.elf* file after launch gdb by using *file* command.

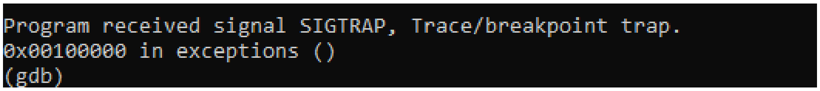
9) Then, connect to the XSDB by running '*target remote localhost:3000'* in the GDB command prompt. Port number 3000 can be found by running a *'netstat -a'*. By default, 3000 is used for PS *ARM* processor. 3001 is for *ARM64* and 3002 is for *Microblaze.* You should see following messages.



```
c:\dotori\prj\slos\slos-git>arm-none-eabi-gdb out/kernel/kernel.elf
GNU gdb (Linaro GDB 2018.06) 8.0.50.20171128-git
Copyright (C) 2017 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:3000
Remote debugging using localhost:3000
0x00000000 in ?? ()
(gdb)
```

Figure 3-11: GDB breaks at 0x0

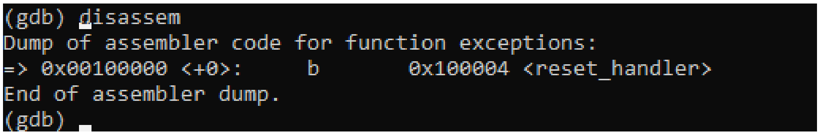
10) Run 'c' to continue in the GDB command console. Since we've alreadyadded a hardware breakpoint at the address 0x100000, CPU 0 will stop like below screen. Excellent!



```
Program received signal SIGTRAP, Trace/breakpoint trap.
0x00100000 in exceptions ()
(gdb)
```

Figure 3-12: GDB breaks at 0x100000 which is the start of the reset exception vector

11) Now, we can debug the secondary bootloader by steppint into it. First run a *disassem* command to see the assembler code. Since we don't have any other exceptioin vector handlers, it should look like below. It makes the flow jump to the reset vector handler which comes right after the reset vector.

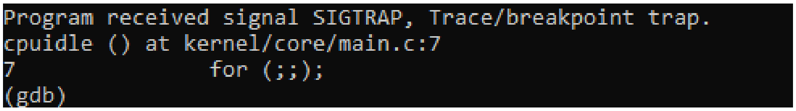


```
(gdb) disassem
Dump of assembler code for function exceptions:
=> 0x00100000 <+0>:     b       0x100004 <reset_handler>
End of assembler dump.
(gdb)
```

Figure 3-13: GDB disassembles at 0x100000

12) Continue by running ‘c’ command in the GDB window. Then, the program counter just jump to the hellowworld reset vector handler that doesn’t do nothing but print ‘helloworld’ message.

If everything goes well, the processor stops at infinite spinning loop and you should see the message in figure 3-7. You can check this by ctrl+c in the GDB window. GDB will break the CPU 0 and shows below screen shot.



```
Program received signal SIGTRAP, Trace/breakpoint trap.
cpuidle () at kernel/core/main.c:7
7               for (;;);
(gdb)
```

Figure 3-14: CPU 0 spinning forever after printing the helloworld message

You can create a *.gdbinit* file storing the repetitive command of GDB configurations. This configuration file is automatically used by the the *arm-none-eabi-gdb* when it starts up. If there is a *.gdbinit* file in the same path as the *arm-none-eabi-gdb* binary, GDB will auto-configure with the command list in *.gdbinit*. If *.gdbinit* file is not in the same directory as a *arm-none-eabi-gdb* binary, ‘*-x path_to_gdbinitfile’* option is necessary to designate the path of configuration file. Open a file with a text editor and add ‘*target remote localhost:3000’ and ‘b main’* to the file and save it as gdbinit.txt. When you run GDB, you need to designate the configuration file path as below example command. The GDB debugger will automatically connect to XSDB debugger.

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

3.8.2 XSDB and GDB Commands

As the example covered in the previous chatper, XSDB commands and GDB commands are used together. Each of them has its own merits; XSDB commands are used to debug lower level such as system level registers. GDB commands are used to more upper level by browsing the kernel source tree. Mostly, we are supposed to use GDB commands, but XSDB commands still plays an important role such as control processor running state or adding hw break points. XSDB also does important role in SMP SLOS debugging later.

A summarized list for frequently used XSDB commands and GDB commands follows.

1) XSDB commands

CMD	Description	Example Usage
bplist	Lists break points.	bpl

bpremove	Removes break point(s).	bpr bp_id bpr all
bpadd	Adds a break point.	bpa 0x400
bpstatus	Shows break point bp_id status	bpstatus bp_id
bpenable	Eanble a break point bp_id	bpenable bp_id
bpdisable	Disable a break point bp_id	bpdisable bp_id
con	Continues from current PC or optionally specified address.	con
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
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
state	Displays the current state of processor.	state
stop	Stops the target processor.	stop
stp	Step into a line of source code or optional count number of lines of source code.	stp stp 100
nxt	Step over a line of source code or optional count number of lines of source code.	nxt nxt 100
stpi	Execute a machine instruction or optional count number of machine instructions.	stpi stpi 100
nxti	Step over a machine instruction or optional count number of machine instructions.	nxti nxti 100
stpout	Step out from current function or optional count number of times.	stpout stpout 10
dis	Disassemble current instruction or optional count number of instructions from address.	dis dis pc 100

locals	Get or Set the value of a local variable.	locals
backtrace	stack back trace.	backtrace

2) GDB Commands

a) Start, Stop, Break commands

r(un)	Start your program	run
q(uit)	Exit GDB	quit

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

b) Execution control commands

c	Continue running programing	c
s(tep)	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 [line 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

c) 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
up	Select frame n frames up	up 2
down	Select frame n frames down	down 2
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

d) 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. Optioins: 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 p/x $r13

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
set	set memory value	set {int}0x20000000 = 1 set variable i = 10

e) 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 *add; line line containing addr list file:{num|func} ; list source line num or func	list list – list 10,50 list main.c:main
set substitute-path	Sets the symbol lookup directory path. This can be used to resolve the source tree path isn’t matched with the ELF.	set substitute-path build_src_path current_src_path

3.9 Summary

In this chapter, we created a simple Helloworld SLOS. This Helloworld SLOS is very simple but it is a skeleton structure used through this book. We will modify this version of SLOS and add new features on top of this. The basic build system and source structure are not changed. While developing the Helloworld SLOS, we learned the GNU make, Makefile, a linker script and how to break the processor with XSDB, GDB debugger.

The GNU make builds the target by going through the two steps. In first step, it generates the dependency graph and in second step, it generates the target files based on the recipes. Makefile defines these the dependency and recipes. We discussed basic structure and syntax of Makefile. We developed the Helloworld Makefile and it is used to the build system of SLOS.

The linker script is used to map the sections in the input object files to the output sections. Linker script also defines the memory address of all these output sections. We developed a simple Helloworld linker script which contains basic sections only.

After we had developed the Makefile and linker script, we build the Helloworld SLOS and successfully boot up the target board. Then, we attached our debuggers (GDB and XSDB debugger) to the target board and broke the processor at the reset vector. After this, we can keep going through the source files with GBD debugger and disassemble the machine code for debugging. Using debugger to stop the running at the exception vector is very important to implement coming features such as interrupt, page fault handling and system call.

References

[1] https://sourceware.org/binutils/docs/ld/Scripts.html#Scripts

[2] The GNU linker, Sourcery G++ Lite 2010q1-188, version 2.19.51, Steve Chamberlain, Ian Lance Taylor

[3] BFD Library: https://en.wikipedia.org/wiki/Binary_File_Descriptor_library

[4] ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition,
http://infocenter.arm.com/

[5] GNU Make, GNU make version 4.2, Richard M. Stallman, Roland McGrath, Paul D. Smith

[6] Xilinx User Guide: UG1043 Embedded System Tools Reference Manual

4 Process Management

4.1 Introduction

Helloworld SLOS in chapter 3 doesn't do meaningful work, it just prints "hello world" message into the terminal screen. But Helloworld SLOS is a really good start place since it covers development environment, bootloader, linker script, reset vector handler and a good debugger. We started from Helloworld SLOS and will expand this simple operating system with more advanced features. From this chapter, this book covers the core implementation of SLOS.

Chapter 4 covers the *process management* portion of SLOS. Figure 4-1 shows the top level of SLOS process management blocks implemented in chapter 4. Process management blocks in figure 4-1 together with new software features are described below.

1) Memory map

   This is not shown in the figure 4-1, but we are going to define our first memory map for process management. This memory map uses physical address only. It will define the placement of each task, stack and heap.

2) ARM GIC (General Interrupt Controller)

   We need to support the timer interrupt to allocate CPU time to each task in a fair way. For this, we first develop the device driver for GIC hardware in PS subsystem.

3) Timer hardware and Timer ISR (Interrupt Service Routine)

   Time information is the base ground for process management. SLOS doesn't have a sophisticated timer framework but this timer framework supports the schedulers. For this, we should implement timer hardware in PS subsystem and its interrupt service routine.

4) Timer Framework

   A timer framework stores the timer information of all tasks. Each timer interrupt updates the time information in the timer framework. We should have a well-defined structure to manage this information. To manage the timer information, SLOS timer framework uses a red-black tree.

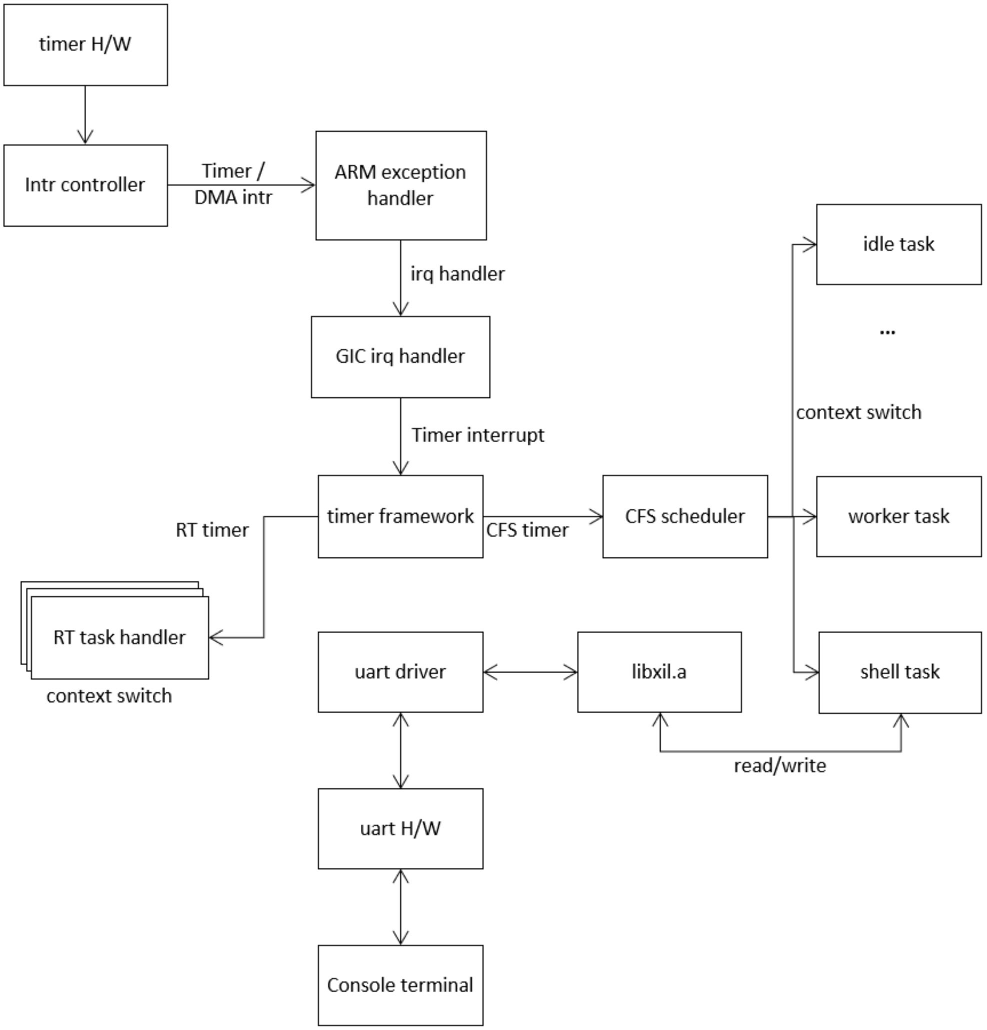


Figure 4-1: SLOS top view when process management is added

5) Scheduler
SLOS process management supports the *CFS (Complete Fair Scheduler)* scheduler and *RT (Real Time)* scheduler. CFS scheduler looks like Linux CFS scheduler. RT scheduler supports the *EDF (Earliest Deadline First)* scheduling scheme.

6) TCB (Task Control Block)
Task control block is the virtualization layer of CPU processor as in the layered model

of process management. We define the SLOS's very simple TCB structure in this chapter.

7) Context Switch
   To support multiple task in a single processor, the context switch is necessary. What context swith does is switching the TCB of current task with the TCB of next task which is picked by the scheduler.

8) forkyi() function
   To fork multiple tasks, SLOS has a forkyi() function. This function creates another TCB and adds new TCB to the runqueue.

9) Heap Management
   While implementing the process management, SLOS needs to allocate some heap memory. This memory is used for task's TCB allocation, timer allocation and so on. The heap manager used in the process management is very simple, even it doesn't support a free() to de-allocate the memory. We will implement a better heap manager in next memory management chapter.

10) Simple RT tasks and CFS tasks for demonstration
    We will create a couple of tasks for process manage demonstration. Especially, the *shell* task plays an important role through this book. It gets user input from console terminal and prints information to the screen. Another special task is *cpu_idle* task. SLOS's *cpu_idle* doesn't do any meaningful work, but normally, it is the place where a power management happens. SLOS doesn't implement the power management but we will cover some theoretical aspect on this. Other tasks are just dummy task for test.

The Helloworld SLOS base sources and basic structures in chapter 3 are still used in chapter 4. But more files are added for process management features. Below are the source tree changes in chapter 4.

1) *kernel/core/gic.c*
   This file is ARM GIC implementations. This file has GIC initialization, GIC irq handler, Interrupt enable/disable mask functions.

2) *kernel/core/ktimer.c*
   This file has the implementations for SLOS timer framework.

3) *kernel/core/main.c*

This file is changed to have new function calls for initialization routines of process management.

4) *kernel/core/ops.S*
This file has an assembler implementation of context switch, IRQ enable/disable masking in CPSR register and *spinlock* implementations.

5) *kernel/core/rbtree.c*
This file has an implmentation of *red-black* tree algorithm. A red-black tree is used to manage a data structure of timers and tasks in a balanced way. This file is directly copied from Linux source.

6) *kernel/core/slmm.c*
This file has a simple heap memory implementation.

7) *kernel/core/task.c*
This file has the implementations for tasks such as forkyi() for a new task creation, context switch, CFS scheduler and other worker task creation.

8) *kernel/core/timer.c*
This file has CPU private timer implementations such as timer enable/disable, timer IRQ, time information update and timer delay function.

9) *kernel/core/wait.c*
This file has task *wait queue* implementations.

These sources are in *SLOS_CH4* branch. You can check out the process management sources by running *'git checkout SLOS_CH4'.*

Again, this book doesn't cover the theoretical discussion about the operating system. This book doesn't describe *what* the process management is. There are tons of good articles and books for this purpose. Rather, this book covers *how* we can implement the concepts of operating system in our own way.

4.2 A Memory Map for Process Management

ARM Cortex-A9 processor is 32bit architecture which can address up to 4GB range. According to reference [1], the external DRAM memory ranges from 0x0004_0000 to 0x3FFF_FFFF. The Zynq7000 evaluaton board has 1GB external memory which is covered by this address range. If you open the block diagram of the default Zynq7000 project in the Vivado which is covered in chapter 2.3.7, you can see this address range in the *Address Editor* window. The rest 3GB address range is predefined as M_AXI_GP0, M_AXI_GP1, I/O peripheral registers, *System Level Control Registers* (*SLCR*), PS system registers, CPU private registers and so on.

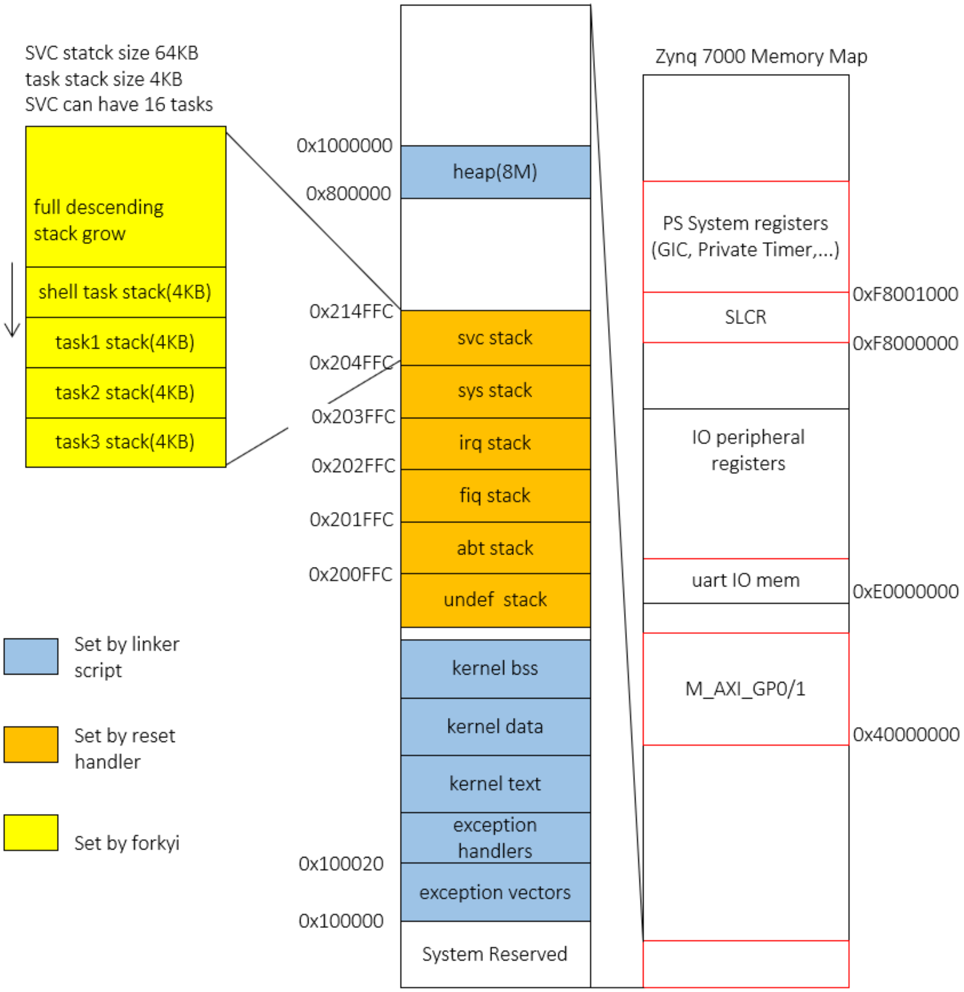


Figure 4-2: A memory map for process management

We can use this 1GB memory but practically, only small portion of this 1GB memory is used for SLOS development. While we are working on the process management, we need a basic memory map defining the placement of each block of process management. Running a process needs to load its code and data into the memory first. It also needs stack for its own purpose; e.g., saving local variables. Some functions need to use a memory that SLOS should provide in the heap memory region. Figure 4-2 shows the memory layout of SLOS for processes management. This memory map will be fully modified in the chapter 5, memory management and continue to be updated through the reset of the chapters.

4.2.1 Kernel Loading and Exception Vectors Addresses

As we already discussed, the kernel loading address is set by the linker script. The linker script of Helloworld SLOS already defined the kernel loading addresses. As described earlier, the linker script has two major roles. Mapping the input sections to output sections and setting up the address of output sections. The *SECTION* body of Helloworld linker script sets the mapping of input sections and the address of these output sections as well.

The *EXCEPTIONS* input section is placed at address 0x0010_0000 and each exception vectors in EXCEPTIONS section have offset 0x4 bytes and there are 8 exception vectors in total. These exception vectors have only one ARM assembler code jumping to corresponding exceptioin handlers. The EXCEPTIOINS section is composed of the exception vectors and its exception handlers.

One thing we need to know at this moment is that we know how to put the EXCEPTIONS input section to the address 0x0010_0000 but the ARM processor still doesn't know where the exception vector is located. The reset handler needs to program the ARM processor about this. We discussed this before but will recall this later again.

Memory map for process management is depicted in figure 4-2. There are three memory address group that need to be set by the SLOS.

1) Memory address set by linker script
   The output sections in the linker script is placed in this memory region. Right after the exception handlers, the SLOS kernel's text code comes. The .text section of output is composed of EXCEPTIONS section and .text section of the rest input files. After the .text section, there comes a .data and .bss sections in a sequential addresses.

2) Memory address set by reset handler
   The reset handler sets the stacks of each ARM processor mode. Each ARM processor mode has a banked *stack pointer (sp)* register and each mode can have its own stack. Normally, this is set in the reset handler.

3) Memory address set by forkyi
   When a task is forked from its parent task, its stack also needs to be set. Since all tasks run in supervisor (SVC) mode in SLOS, they all share the SVC mode stack. SVC mode has 64KB stack size and a task has 4KB stack size, which means SLOS supports up to 16 tasks by design. A task's stack works as a *full-descending* mode. This stack mode operation was explained in chapter 2.5.

In addition, there is a 8MB heap memory used by a memory manager. There are also *memory mapped* peripheral memory regions which are predefined by Zynq7 processor. These are master AXI interface, IO peripherals, System Level Control Registers (SLCR), and PS subsystem registers such as GIC, Timers and so on.

We need to program some registers in the PS subsystem registers later. When doing this, since ARM processor is using memory mapped IO, we can easily access these registers by using their memory address. Since we don't enable the memory management unit (MMU) in current SLOS, all of these addresses are physical address, which means the CPU logical address is same as the memory physical address.

4.2.2 Stack Addresses

Stack address is pointed by the stack pointer (sp or r13) register. Since each processor mode has its own stack, the stack pointer register is a banked register among processor modes as described in table 2-5.

Stack is a memory necessary for the processor to run a program code. This is solely for processor needs and applications doesn't take care of this, even it doesn't know stack is used while running. But the operating system that is placed at the boundary edge of software layers should take care of the stacks of each application or task.

The process management of SLOS should define the stack of each task before running it. In addition, each mode of processor should have a separate stack from other processor mode. So, there are two types of stack in SLOS.

1) Stack for each processor mode
   This stack is necessary for the processor to run in its different modes. When the processor enters any of these modes, the banked stack pointer register is used. When process enters a different mode, the banked registers including stack pointer register are restored. Changing mode of the processor is simply done by changing the Mode bit of the CPSR register.

2) Stack for each task
   All tasks in SLOS run in Supervisor (SVC) mode of the processor. The stack of SVC mode is shared among all tasks. Each task is allocated its own stack by forkyi() to run its own code. As the stack pointer is banked in each processor mode, the stack pointer for each task is also banked in each task's context. When current task is changed to another task, context switch must backup and restore the stack pointer properly.

Memory map in figure 4-2 shows the stack layout in SLOS. The stack in SLOS is full descending mode. How the full descending mode stack works is described in chapter 2.5. Stack uses 4KB of memory except SVC stack. Since there is no boundary checking in stack operation, be careful not to overflow in stack. For example, if a task declares its local array with 4KB size, the stack overflows which results in the crash of operating system. SVC stack size pretty bigger than others; SVC stack size is 64KB. This is because all tasks are running in SVC mode and 64KB stack size can support 16 tasks in SVC mode.

The stack pointer value for processor mode is set at the reset vector handler. The stack pointer values for each task is set at the forkyi () function when a new task is created.

4.2.3 Heap

Heap is a memory region that is reserved for application's needs. Unlike stack which is automatically controlled by the processor after the stack pointer is set, heap is solely for application (including OS) usage and is controlled by the requester after allocated. For example, an application needs a memory to store an image, it asks operating system to serve that memory. Then, the memory manager in operating system allocates that amount of memory in the heap and returns the start address of the heap to that application. After getting the heap address, the application should properly use this memory, should not go over the allocated region and should free after finishing its use. Heap is just one of the resources that operating system manages. When application wants a memory, it should ask the operating system to allocate a heap for it. But after operating system returns the start address of heap allocation, the management of the heap is solely the application's role.

Heap memory is used in many places in SLOS. A task creation in forkyi () function allocate a task control block (TCB) in the heap. Resources in timer framework also needs the heap allocation. For this, there is a very simple heap memory management in the process management. This is a temporary heap management only to meet the memory needs of process management. In chapter 5, memory management will properly implement the virtualization of the heap memory region.

Heap memory in process management has 8MB size starting from 0x0080_0000 to 0x0100_0000. This size and region are also changed in chapter 5. See below kmalloc() function implementation. It gets the size of memory requirement and returns the start address of allocated heap region. It has a static variable *static uint8_t *heap* pointer to keep track of the start address of empty heap region. This means the kmalloc() function is not *thread-safe:* if multiple threads calls the kmalloc() simultaneously, the vairable *heap* pointer could be corrupted. Even though this kmalloc() function is not *thread-safe,*that doesn't matter for this temporary heap memory management. The kmalloc() function does two things.

1) increment the heap pointer by size amount in line 14.
2) assign the *prev_heap* pointer with the start address of free heap and return the

prev_heap value.

```
void *kmalloc(uint32_t size)
{
    static uint8_t *heap = NULL;
    uint8_t *prev_heap;

    if (!heap) {
        heap = (uint8_t *)(&__kernel_heap_start__);
    }

    prev_heap = heap;
    if ((int)(heap + size) >= (int)(&__kernel_heap_end__)) {
        return 0;
    }
    heap += size;
    return (void *) prev_heap;
}
```

The *__kernel_heap_start__ and __kernel_heap_end__* are set in the linker script. This temporary slmm doesn't have an implementation of kfree() function.

4.2.4 Other Memory Regions

As figure 4-2 shows, only very small portion of 1GB memory are used for SLOS process management. Other regions such as AXI bus master, I/O peripheral registers start from 0x4000_0000. These regions are predefined and the Zynq FSBL bootloader initialize them. The UART registers used for printing "hello world" message is also placed at the I/O peripheral registers as shown in figure 4-2. Other important registers are System Level Control Registers (SLCR), PS subsystem registers, CPU private registers. The usage of those registers are added as the implementations of SLOS advances through this book. For a detailed description of these memory region, refer the Zynq7000 technical reference document [1].

Since these registers are system critical information and settings, they can be read and be written only in priviledged mode of processor. Any unsecure user applications are not allowed to directly access to them. Only operating system can access them and expose the information to user applications in a secure way. That's why the operating system runs in privileged mode and user application runs in USR mode of unprivileged processor mode.

4.3 Reset Handler and IRQ Handler

The exception handling is a center of SLOS blocks as in the top view in chapter 1. It connects diverse exceptions to associated exception handler routines. These exceptions occur while current software is running. It interrupts current program, generates faults while accessing memory, becomes a channel from user space to kernel space and so on. If there is no exception in the processor, there is no way to kick out current running program. In SLOS process management chapter, we first modify the reset handler in Helloworld SLOS, then implement the timer interrupt in coming chapters 4.4, 4.5.

The reset handler for process management will have more meaningful works from the list in chapter 3.4 such as vector base address initialization, processor mode stack initialization. More initialization will be added while implementing new features.

4.3.1 Exception Handling

Since this is the first time to implement a complete reset handler and IRQ handler in SLOS, let's first dig into the exception handling in ARM. As in chapter 2.5.4, ARM Processor Modes, there are exception handler, and each of them is associated with the ARM processor mode with the same name. Each exeption handler processes a corresponding ARM process mode.

When exeption occurs while a program is running, entering / returning from exception handler looks like figure 4-3. Exception interrupts current program running whenever it needs to.

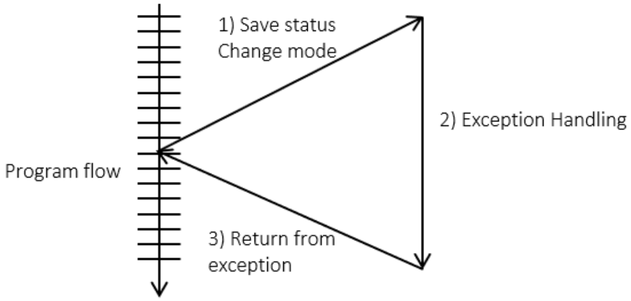


Figure 4-3: Exception flow while a program running

In step 1) in figure 4-3, when an exception occurs, ARM core automatically does the following before entering an exception handler [9]:

1) Copies the CPSR to the SPSR_<mode>, the banked register specific to the (non-user) mode of operation.

2) Stores a return address in the Link Register (LR) of the new mode.
3) Modifies the CPSR mode bits to a mode associated with the exception type.
4) Sets the PC to point to the relevant instruction from the exception vector table.

When in the new mode, the core will access the register associated with that mode. It will almost always be necessary for the exception handler software to save registers onto the stack immediately on exception entry.

In step 2), to return from an exception handler, two separate operations must take place atomically:

1) Restore the CPSR from the saved SPSR.
2) Set the PC to the return address offset.

There are a number of ways to achieve this. You can use a data processing instruction to adjust and copy the LR into the PC, for example:

```
SUBS pc, lr, #4
```

Specifying the '*S*' means the SPSR is copied back to the CPSR at the same time. If the exception handler entry code uses the stack to store registers that must be preserved while it handles the exception, it can return using a load multiple instruction with the '^' qualifier. For example, an exception handler can return in one instruction using:

```
LDMFD sp! {pc}^
LDMFD sp!,{R0-R12,pc}^
```

The ^ qualifier in this example means the SPSR is copied to the CPSR at the same time. To do this, the exception handler must save the following onto the stack:

1) All the work registers in use when the handler is invoked.
2) The link register, modified to produce the same effect as the data processing instructions.

Since SLOS has only SVC mode for its operation, backup / restore of CPSR isn't necessary. As you can see the real implementation of Reset handler and IRQ handler in the chapter 4.3.3, entering / exiting of exception handler is quite straightforward. As we progress in other subsystem's implementation, more exception handler will be added.

4.3.2 System Level Interrupt

Figure 4-4 is the system level block diagram of Zynq-7000 interrupt subsystems. The *Generic Interrupt Controller (GIC)* is the interface hardware between interrupt source and the CPU processors. GIC does following works:

1) Enable / Disable interrupt.
2) Classify interrupts.
3) Prioritize interrupts.
4) Distribute interrupts.

A detailed implementation of GIC will be covered in chapter 4.4. This chapter explains the high-level descriptions of Zynq7000 interrupts. Notice that there can be up to 16 interrupts coming from PL subsystem. The GIC is placed in-between these interrupt signals and the CPU processors. All these interrupt signals go into the GIC and the GIC distributes the interrupt to proper CPU processor.

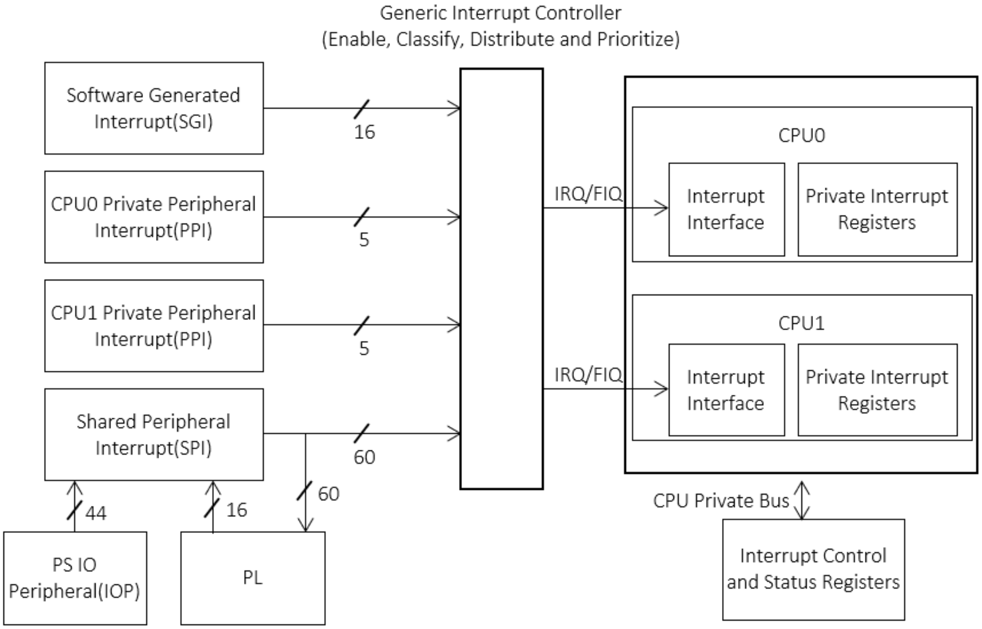

Generic Interrupt Controller
(Enable, Classify, Distribute and Prioritize)
Software Generated Interrupt(SGI)
16
CPU0 Private Peripheral Interrupt(PPI)
5
CPU1 Private Peripheral Interrupt(PPI)
5
Shared Peripheral Interrupt(SPI)
60
44
16
60
PS IO Peripheral(IOP)
PL
IRQ/FIQ
IRQ/FIQ
CPU0
Interrupt Interface
Private Interrupt Registers
CPU1
Interrupt Interface
Private Interrupt Registers
CPU Private Bus
Interrupt Control and Status Registers

Figure 4-4: Zynq7000 system level interrupt

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

1) *Software Generated Interrupt (SGI)*
   This is an interrupt generated by software writing SGI interrupt number to an *ICDSGIR* register and specify the target CPU(s). Zynq-7000 has 16 SGI interrupts from interrupt 0 to 15. Normally, the system uses this interrupt for *interprocessor communication (IPC).* In SLOS implementation, we don’t use this interrupt.

2) *Private Peripheral Interrupt (PPI)*
   This 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 as in figure 4-4. Zynq7000 PPI interrupts are described in table 4-1. Those are Global Timer, IRQ, FIQ, private timer, and watchdog timer. There are also reserved PPI interrupts. SLOS uses CPU0 private timer for its timer framework implementation.

3) *Shared Peripheral Interrupt (SPI)*
   This is a peripheral interrupt that the *GIC Distributor* can route to any of specified combination of processor. Zynq7000 has 16 PL subsystem interrupts which are shared peripheral interruts ranging from IRQ ID #61 ~ #68, #84 ~ #91. SLOS will use one of these PL subsystem interrupts in Hardware-Software codesign.

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: Private Peripheral Interrupts (PPI)

The unique, single interrupt that SLOS needs for its process management is a private timer interrupt. This interrupt is a core, heart beat to manage and schedule all tasks. SLOS uses this interrupt to implement the timer framework and the scheduler for process management.

4.3.3 Reset Handler and IRQ Handler

The Helloworld SLOS in chapter 3 has only a simple reset vector in the exception vector. All other exception vectors are not defined in chapter 3. In this chapter, we are going to add more implementations into the reset handler and add the irq handler into the exception vector. In the SLOS Git repository, run below command to check out a branch for chapter 4.

git checkout SLOS_CH4

The branch SLOS_CH4 has the implementation of process management. This book will use SLOS_CH4 branch for chapter 4.

Since the exception handlers are the center of SLOS, let’s look into the exception handler implementations in more detail. Below implementation is the content of kernel.S.

```
#include "mem_layout.h"

/* arm exception code */
.set MODE_SVC, 0x13
.set MODE_ABT, 0x17
.set MODE_UND, 0x1b
.set MODE_SYS, 0x1f
.set MODE_FIQ, 0x11
.set MODE_IRQ, 0x12
.set I_BIT, 0x80
.set F_BIT, 0x40
.set IF_BIT, 0xC0
.set CONTEXT_MEM, 0x4000
.set SP_MEM, 0x4100

.extern platform_undefined_handler
.extern platform_syscall_handler
.extern platform_prefetch_abort_handler
.extern platform_data_abort_handler
.extern gic_irq_handler
.extern platform_fiq_handler

.extern main
.section EXCEPTIONS, "ax"
.arm
.global exceptions

exceptions :
        b reset_handler
        b undefined_instruction_handler
        b syscall_handler
        b prefetch_abort_handler
        b data_abort_handler
        b reserved
        b irq_handler
        b fiq_handler

reset_handler:
         /*set VBAR with 0x100000 */
        ldr         r0, =0x100000
        mcr         p15,0,r0,c12,c0,0

         /*change to supervisor*/
        msr         CPSR_c, #MODE_SVC | I_BIT | F_BIT
        /*; setup svc stack*/
        ldr         r0,=SVC_STACK_BASE
        mov         r13, r0

        /*; Switch to undefined mode and setup the undefined mode stack*/
        msr    CPSR_c, #MODE_UND | I_BIT | F_BIT
        ldr         r0,=UNDEF_STACK_BASE
        mov     r13, r0

        /*; Switch to abort mode and setup the abort mode stack*/
        msr    CPSR_c, #MODE_ABT | I_BIT | F_BIT
        ldr         r0,=ABT_STACK_BASE
        mov     r13, r0

        /*; Switch to SYS mode and setup the SYS mode stack*/
        msr    CPSR_c, #MODE_SYS | I_BIT | F_BIT
        ldr         r0,=SYS_STACK_BASE
        mov     r13, r0

        /*; Switch to IRQ mode and setup the IRQ mode stack*/
        msr    CPSR_c, #MODE_IRQ | I_BIT | F_BIT
        ldr         r0,=IRQ_STACK_BASE
        mov     r13, r0

        /*; Switch to FIQ mode and setup the FIQ mode stack*/
        msr    CPSR_c, #MODE_FIQ | I_BIT | F_BIT
        ldr         r0,=FIQ_STACK_BASE
        mov   r13, r0

        /*; Return to supervisor mode*/
        msr   CPSR_c, #MODE_SVC

        /*; jump to main */
        b           main
;
undefined_instruction_handler:
        b           platform_undefined_handler
;
syscall_handler:
        bl          platform_syscall_handler
;
prefetch_abort_handler:
        b           platform_prefetch_abort_handler
;
data_abort_handler:
        bl          platform_data_abort_handler
;
reserved:
        b           .
;
irq_handler:
        stmia       r13, {r4-r6}
        mov         r4, r13
        sub         r5, lr, #4
        msr         cpsr_c,#MODE_SVC | I_BIT | F_BIT /* irq/fiq disabled, SVC mode */
        mov         r6,#CONTEXT_MEM
        str         r5, [r6],#4 /* save return addr */
        str         lr, [r6],#4  /* save current task lr */
        str         r13, [r6],#4 /* save sp */
        stmia       r6!, {r0-r3}
        mov         r1, r6
        ldmia       r4, {r4-r6} /* restore r4-r6 */
        stmia       r1!,{r4-r12}
```

```
108            mrs         r5, spsr
109            str         r5, [r1],#4 /* save spsr */
110            bl          gic_irq_handler
111     ;
112            mov         r12, #CONTEXT_MEM
113            add         r12,r12,#4
114            ldr         r14,[r12],#4
115            ldr         r13,[r12],#4
116            ldmia       r12!,{r0-r11}
117            add         r12,r12,#4
118            ldr         r12,[r12]
119            msr         cpsr_c,#MODE_SVC
120            mov         r12,#CONTEXT_MEM
121            ldr         r12,[r12]
122            mov         pc,r12
123     ;
124
125     fiq_handler:
126            bl          platform_fiq_handler
127     ;
128     .end
```

kernel.S for process management has two exception vector implementations; reset handler and irq handler. We already covered the ARM exception vectors in chapter 2.5. Each exception vector which is placed from address 0x0010_0000 has a branch command to the proper handler routines. These exception vectors and their handlers are built into the EXCEPTIONS section.

There are 8 exception vectors in SLOS which is placed from line 29 to 36.

1) reset_handler
   The entry point of SLOS. We added simple reset vector handler in Helloworld SLOS. We will modify the previous implmentation. The reset_handler implementation of process management doesn't change once we implement it in this chapter. The offset of reset vector is 0x0 from base address.

2) undefined_instruction_handler
   Not used. Handler just spins forever. Offset is 0x4 from base address.

3) syscall_handler
   System call handler. The implementation for this exception handler will be added in chapter 6. Handler just spins forever for now. Offset is 0x8 from base address.

4) prefetch_abort_handler
   Not used. Handler just spins forever. Offset is 0xC from base address.

5) data_abort_handler
   This exception handler will be used in chapter 5 memory management. Currently, handler just spins forever. Offset is 0x10 from base address.

6) reserved
   Not used. Offset is 0x14 from base address.

7) irq_handler
   The interrupt handler implementation. For process management, there is timer irq handler will be implemented. Chapter 7 will add another 2 interrupts to handle custom hardware event. Offset is 0x18 from base address.

8) fiq_handler
   Not used. Handler just spins forever. Offset is 0x1C from base address.

Below table 4- 2 is the summary of these exception vectors.

Exception	Offset from vector base	Mode on entry	F bit on entry	I bit on entry	Action
reset	0x00	supervisor	disabled	disabled	branch its handler routine
undefined instruction	0x04	undefined	unchanged	disabled	
software interrupt	0x08	supervisor	unchanged	disabled	
prefetch abort	0x0C	abort	unchanged	disabled	
data abort	0x10	abort	unchanged	disabled	
reserved	0x14	reserved	-	-	
IRQ	0x18	irq	unchanged	disabled	
FIQ	0x1C	fiq	disabled	disabled	

Table 4-2: Summary on exception vectors

The *.section* keyword in line 24 designates the EXCEPTION input section which is used for section mapping in linker script. 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 a syscall handler for the system call implementation and data abort handler for page translation fault in the future. Only reset handler and irq handler are needed for process management.

4.3.3.1 Reset Handler

Reset handler has offset 0 from the base of exception vector which is 0x0010_0000 in SLOS and has the first, one-time executed instructions. This entry address is set by linker script with *ENTRY* command. Reset handler of SLOS does two important things; those are 1) setting *VBAR (exception Vector Base Address Register)* and 2) setting the stack address of each processor mode. Line 40, 41 are for setting the VBAR which is 0x0010_0000. As the name represents, the VBAR register provides the base address for exceptions. 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
```

The *MRC, MCR* instructions are used to move data between general purpose register and coprocessor register. A coprocessor is a processor used to supplement the functions of the primary processor (the ARM Cortex-A9 in Zynq7000). Operations performed by the coprocessor could be floating point arithmetic, graphics, signal processing, encryption or I/O interfacing with peripheral devices. By offloading processor-intensive tasks from the main processor, coprocessors can accelerate the whole system performance [1]. ARM can have 16 coprocessors which are from CP0 to *CP15*. *CP10, CP11, CP14* and *CP15* coprocessor numbers are reserved as below:

CP10: Vector Floating Point (VFP) coprocessor control
CP11: Vector Floating Point (VFP) coprocessor control
CP14: Debug and ETM coprocessor control
CP15: System coprocessor control

CP15 coprocessor plays important role in virtual memory management in next chapter. Zynq7000 is a really good chipset in the sense that the PL subsystem can be programmed as a custom coprocessor to perform specific tasks. We will design a custom coprocessor hardware in chapter 7. There is a full list of CP15 registers in section B3.17.2 in ARM architecture reference document [3].

The other thing done in reset handler is setting the stack address of each processor mode. We learned that the general-purpose register r13 (sp, stack pointer) is a banked register of each processor mode; each mode has its own stack address. These stack addresses are set in the reset handler. Line 43~73 are the routines to set the stack address of each processor mode. First, it changes the processor mode by setting the CPSR register's mode bits and set the stack pointer (r13) with associated address value which is designed in the memory map in figure 4-2. Refer figure 4-2 for the detailed memory map for process management.

Each mode in SLOS has 4KB stack size except supervisor mode. Supervisor mode has 64KB stack which is shared through all kernel tasks. Each task running in SVC mode has also 4KB stack size. So, supervisor mode can have 16 tasks in total. Since SLOS always runs in supervisor mode, the maximum number of kernel task is 16. Nonetheless, the 16 max task is enough for this simple, hobby SLOS. If you still don't like this limitation, you can increase this number simply by redesign the memory map.

After finishing all these, the reset handler jumps to kernel's main() entry function in line 79.

4.3.3.2 IRQ Handler in SLOS

An IRQ exception is raised by external hardware. When IRQ signal is raised, the core performs several steps automatically. The CPSR register is copied to SPSR of current mode. This saves the current mode, interrupt mask, and condition flags. Then, processor mode is changed to a appropriate mode. The CPSR content is updated so that the mode bits reflect the IRQ mode, and the I bit is set to mask additional IRQs. The contents of the PC (Program Counter) in the current execution mode are stored in LR_IRQ. The PC is set to the IRQ entry in the vector table. After these steps are done by the processor, SLOS interrupt handler implementation starts.

The priority and list of cores to which an interrupt can be delivered to are all configured in the *interrupt distributor*. An interrupt asserted to the distributor by a peripheral will be marked in *Pending* state (or *Active and Pending* if was already *Active*). The distributor determines the highest priority pending interrupt that can be delivered to a core and forwards that to the *CPU interface* of the core. At the CPU interface, the interrupt is in turn signalled to the core, at which point the core takes the FIQ or IRQ exception [9].

The core executes the exception handler in response. The handler must query the interrupt ID from a CPU interface register and begin servicing the interrupt source. When finished, the handler must write to a CPU interface register to report the end of processing. Later on, the CPU interface is prepared to signal the next interrupt forwarded to it by the distributor.

While servicing an interrupt, the distributor cycles through Pending, Active states, ending in Inactive state when it has finished. The state of an interrupt is therefore reflected in the distributor registers.

Line 96 to line 123 in the kernel.S file are SLOS IRQ handler implementation. SLOS IRQ handler has 3 parts of implementation. Register saving routine, IRQ service routine, and register restoring routine.

1) Saving Register

This is done in line 100 ~ 109. SLOS changes its mode to supervisor mode and saves all of processor registers into *CONTEXT_MEM* area located at address 0x0000_4000. The processor registers are saved like figure 4-5. Actually, this is not an optimal way to save and restore context. Using the load/store multiple described in chapter 2.5 is a better way to save and store registers.

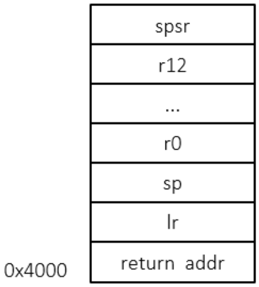


Figure 4-5: CONTEXT_MEM after saving current task's context

Saving processor registers means taking a snapshot of current task's context. Task context is saved into Task Control Block (TCB) that is a virtualization of the processor. This also prepares an upcoming *context switch* which occurs based on the scheduler's decision.

2) IRQ Service

This is the second part in IRQ handler which is jumping to a specific *ISR (IRQ Service Routine).* IRQ service routine is associated with an interrupt service function which is pre-registered into the interrupt service routine vector. Interrupt service routine vector is an array of pointers to a function that implements the logic of interrupt service. The interrupt index number is used to this array entry index. This is happening in line 110, gic_irq_handler() function.

3) Restoring Register

This is the return part of IRQ handler. This part restores all saved registers (a snapshot of current task's context) to the processor's registers and goes back to the interrupted location. In IRQ handler, the return address is *lr - 4* as in line 98. This is because of a pipelined operation of ARM ISA. Pipelined operation is beyond this book and just keep it mind that the return address of IRQ handler is lr - 4.

Figure 4-6 is a flow diagram of IRQ handler implementation. This flow becomes the basic part of context switching. As in figure 4-6, the context switching happens simply by changing the content of *CONTEXT_MEM* with the next task's context. The scheduler selects a proper next task. We will cover the IRQ handler routine in detail along with context switch in chapter 4.6.4. In this chapter, just notice that IRQ handler can be implemented with the upper 3 parts and the preparation of mode changes. The ISR is a specific function associated with the interrupt service logic. We are going to add additional two more ISRs at the end of this book, but process management needs only one ISR - timer ISR.

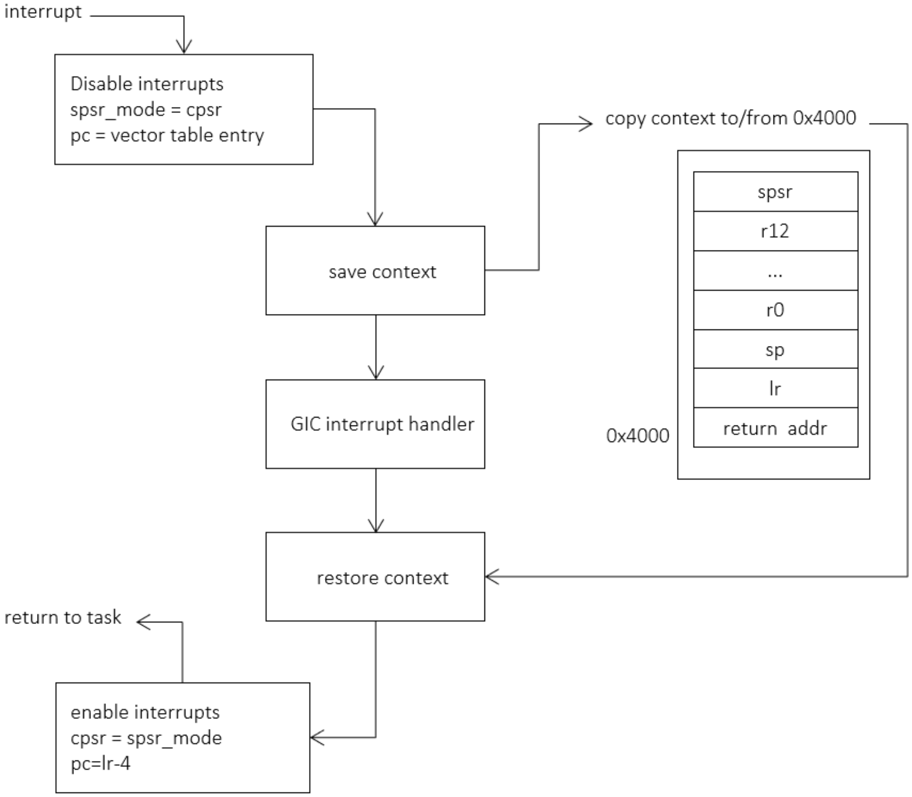


Figure 4-6: IRQ handler flow

4.4 Generic Interrupt Controller (GIC) Implementation

As we are working on the operating system development, the lowest edge of software in the layered model (see figure 1-1), we need to not only develop software implementation of IRQ handler, but we also need to implement the hardware driver of the interrupt. Operating system should be able to handle its system resources. There is a hardware block that handles the interrupt signal in the lowest level. ARM defines the *Generic Interrupt Controller* (*GIC*) architecture [4] for this purpose such as

1) the architectural requirement for handling all interrupt sources for any processor connected to GIC
2) a common interrupt controller programming interface applied to uniprocessor or multiprocessor systems.

The GIC is a centralized resource for supporting and managing interrupts in a system that includes at least one processor. The GIC provides registers for managing interrupt sources, interrupt behavior, and interrupt routing to one or more processors. It supports for:

1) enabling, disabling, and generating processor interrupts from hardware (peripheral) interrupt sources,
2) Software-generated Interrupts (SGI),
3) interrupt masking and prioritization,
4) wakeup events in power management environments.

In addition to these basic abilities, GIC supports for Interrupt Grouping, ARM architecture extensions like Virtualization Extensions, Security Extensions and other features. Xilinx Zynq7000 chipset adopted ARM GIC version 1.0 which is a little bit old. There is also a version 2.0 GIC specification.

4.4.1 GIC Partitioning

SLOS needs very limited features of GIC. Although we want to stick to the simplicity of operating system, at least we need to know two things about GIC*; Distributor and CPU Interfaces*. The GIC architecture splits into a Distributor block and one or more CPU interface blocks.

1) Distributor Block

This block does interrupt prioritization and distribution to the next CPU interface blocks that are connected to the processors. It centralizes all interrupt sources, determines the priority of each interrupt and forwards the interrupt with highest priority to the interface for each CPU interface, does interrupt masking and preemption handling. The Distributor block identifies each interrupt with ID numbers. The interrupt ID 0~31 are private to each CPU. These interrupts are *banked* in the Distributor. The Distributor register block registers are identified by the ICDprefix.

The distributor hosts a number of registers that you can use to configure the properties of individual interrupts. These configurable properties are [9]:

a) Interrupt Priority

The distributor uses this to determine which interrupt is next forwarded to the CPU interface.

b) Interrupt Configuration
This determines if an interrupt is level- or edge-sensitive.

c) Interrupt Target
This determines a list of cores to which an interrupt can be forwarded.

d) Interrupt Enable or Disable Status
Only those interrupts that are enabled in the distributor are eligible to be forwarded when they become pending.

e) Interrupt Security
This determines whether the interrupt is allocated to Secure or Normal world software.

f) Interrupt State
Each interrupt has its state and this state is updated while it is handled. This is covered in next chapter.

The distributor also provides priority masking by which interrupts below a certain priority are prevented from reaching the core. The distributor uses this when determining whether a pending interrupt can be forwarded to a particular core.

2) *CPU Interface Blocks*
This block does priority masking and preemption handling of a connected processor. CPU interface block mostly enables the signaling of interrupt requests to the processor, acknowledging an interrupt, indicating completion of the processing of an interrupt. It is important to write to the CPU interface to indicate the interrupt completion when the handler on the processor has completed the processing of an interrupt. CPU interface block registers are identified by the *ICCprefix*.

Figure 4-7 shows the GIC's Distributor blocks and CPU interface blocks. Notice that there are only 5 PPIs used in Zynq7000, which is described in table 4-1. Also notice that the PPI and SGI interrupts are banked to each CPU and have the same interrupt IDs. The PPI and SPI interrupts are coming from out of the CPU but the SGI interrupt comes from inside of a CPU. Software can use the SGI interrupt to send a signal to other core processors in multicore system. We don't use this.

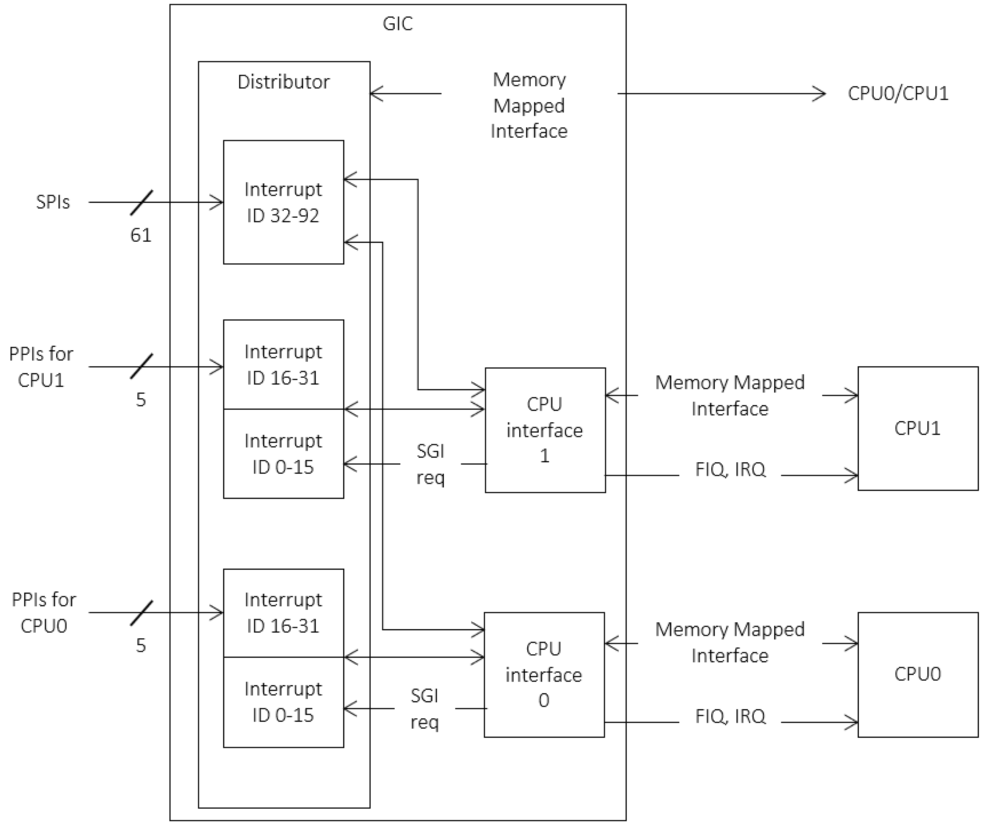


Figure 4-7: GIC component blocks

4.4.2 Interrupt State

The GIC Distributor maintains a state machine for each supported interrupt on each CPU interface. There are 4 possible states of interrupt as below.

1) Inactive
   An interrupt that is not active or pending.

2) 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.

3) Active
   An interrupt from a source to the GIC that has been acknowledged by a processor, and is being serviced but has not completed.

4) Active and Pending
   A processor is servicing the interrupt and the GIC has a pending interrupt from the same source.

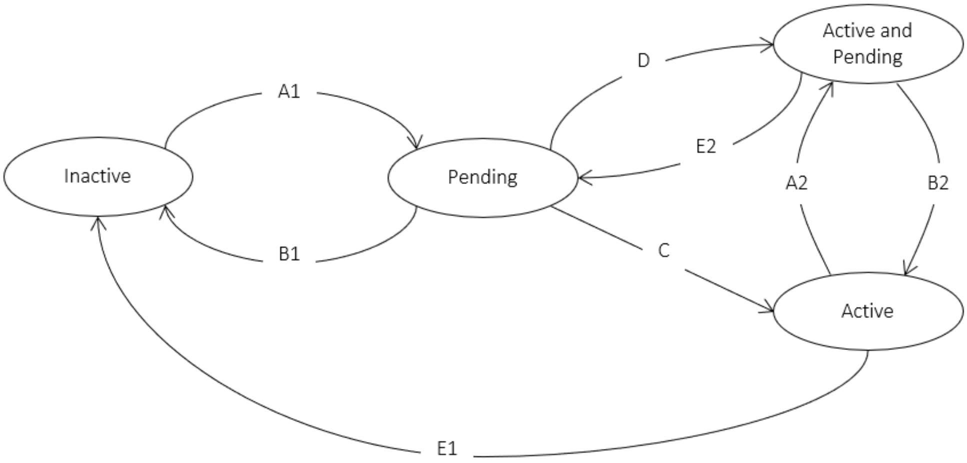


Figure 4-8: Interrupt state machine and state transition

Figure 4-8 shows the interrupt state machine and its state transitions. Let's look into more detail on the state transitions. This transition is applied to PPI and SPI interrupts. For SGI interrupt and for more details, refer the document [4].

1) Transition A1 or A2, add pending state
   When a peripheral hardware asserts an interrupt request signal, or software writes to an ICDISPRn (*Interrupt Set-Pending Register n*).

2) Transition B1 or B2, remove pending state
   Either when the signal is deasserted in level sensitive interrupt or when the ICDICPRn *(Interrupt Clear-Pending Register n)* is written in edge-triggered interrupt

3) Transition C, pending to active
   When software reads ICCIAR *(Interrupt Acknowledge Register)* if the interrupt is enabled and has sufficient priority. Software can read the ICCIAR register to obtain the interrupt ID.

4) Transition D, pending to active and pending
   When both conditions that software reads ICCIAR register and interrupt signal is valid are met. Reading ICCIAR register adds pending state. In addition, the level-

trigger signal still remains in level-sensitive interrupt or another edge-trigger signal occurs.

5) Transition E1 or E2, remove active state
When software deactivates interrupt by writing to either ICCEOIR *(End Of Interrupt Register).*

This interrupt state diagram and its state transition are deeply related to the implementation of interrupt handler. This information is used when we implement our timer interrupt.

4.4.3 Interrupt Handling Sequence

Before digging into the GIC impementation, let's be clearer on the sequence of the whole interrupt handling including GIC part. This is from reference [9]. When the core takes an interrupt, it jumps to the top-level interrupt vector obtained from the vector table and begins execution. The top-level interrupt handler reads the Interrupt Acknowledge Register from the CPU Interface block to obtain the interrupt ID. As well as returning the interrupt ID, the read causes the interrupt to be marked as active in the distributor. Once the interrupt ID is known (identifying the interrupt source), the top-level handler can now dispatch a device-specific handler to service the interrupt.

When the device-specific handler finishes execution, the top-level handler writes the same interrupt ID to the End of Interrupt register in the CPU Interface block, indicating the end of interrupt processing. Apart from removing the active status, which will make the final interrupt status either Inactive, or Pending (if the state was Active and Pending), this will enable the CPU Interface to forward more pending interrupts to the core. This concludes the processing of a single interrupt.

It is possible that there are more than one interrupt waiting to be serviced on the same core, but the CPU Interface can signal only one interrupt at a time. The top-level interrupt handler repeats the above sequence until it reads the special interrupt ID value 1023, indicating that there are no more interrupts pending at this core. This special interrupt ID is called the *spurious interrupt* ID.

The spurious interrupt ID is a reserved value, and cannot be assigned to any device in the system. When the top-level handler has read the spurious interrupt ID it can complete its execution, and prepare the core to resume the task it was doing before taking the interrupt.

In summary, the interrupt signal is treated in following sequences.

1) The interrupt signal is asserted from interrupt source hardware or software running in other core.

2) The GIC detects the interrupt signal and determines the interrupts that are enabled. An interrupt that is not enabled has no effect on the GIC.

3) For each pending interrupt, the GIC determines target the processor or processors.

4) The GIC Distributor forwards the highest priority pending interrupt to the targeted CPU interface.

5) Each CPU interface determines whether to signal an interrupt request to its processor, and if required, does so.

6) The processor goes to IRQ mode and PC jumps to the IRQ handler.

7) IRQ handler saves the current interrupted context and return address.

8) Software runs the interrupt handler. It must do below two things.
   a) Before running a specific *Interrupt Service Routine (ISR)* function, the IRQ handler acknowledges the interrupt by setting ICCIAR register and the GIC returns the interrupt ID and update the interrupt state to active.
   b) Calls an ISR associated with the current interrupt ID.
   c) After processing the interrupt, the IRQ handler signals *End of Interrupt* (*EOI*) to the GIC by setting ICCEOIR register.

9) IRQ handler restores the task context which was interrupted by the peripheral hardware and the software reruns at the right place by using saved return address.

The steps 6) ~ 8) is described in figure 4-6 which is corresponding to the SLOS IRQ handler implementation.

4.4.4 GIC Implementation

Interrupt initialization is referenced from [9]. Both the distributor and the CPU interfaces are disabled at reset. The GIC must be initialized after reset before it can deliver interrupts to the core. In the distributor, software must configure the priority, target, security and enable individual interrupts. The distributor block must subsequently be enabled through its control register. For each CPU interface, software must program the priority mask and preemption settings. Each CPU interface block itself must be enabled through its control register. This prepares the GIC to deliver interrupts to the core. Before interrupts are expected in the core, software prepares the core to take interrupts by setting a valid interrupt vector in the vector table, and clearing interrupt masks bits in the CPSR.

The entire interrupt mechanism in the system can be disabled by disabling the distributor block. Interrupt delivery to an individual core can be disabled by disabling its CPU interface block, or by setting mask bits in CPSR of that core. Individual interrupts can also be disabled (or enabled) in the distributor. For an interrupt to reach the core, the individual interrupt, distributor and CPU interface must all be enabled, and the CPSR interrupt mask bits cleared.

GIC initialization routines in SLOS are in *kernel/core/gic.c* file. The GIC initialization is done in gic_init() function and it is called from kernel main entry function when kernel boots up. The gic_init() function initializes the GIC Distributor and CPU interfaces. The Distributor initialization is done through below steps. The Distributor implementation is in init_gic_dist() fucntion.

1) Disabling GIC by writing *'0'* to ICDDCR (Distributor Control Register) register.

2) Configure interrupt sensitivity by using ICDICRn (Interrupt Configuration Register) registers.

3) Set interrupt target cpu with CPU0 by using ICDIPTRn (Interrupt Processor Target Register) registers.
4) Disable all interrupts by using ICDICERn registers (Interrupt Clear Enable Register n).

5) Enable GIC by writing '1' to ICDDCR (Distributor Control register).

The CPU Interface initialization is done by the following steps. The CPU Interface is in *init_gic_cpu()* function.

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 IRQ handler vector to implement each interrupt service logic. It is just a fuction pointer and is defined in *kernel/inc/gic.h*. Each entry of this vector table is associated with a specific interrupt and indexed with the interrupt ID. The structure definition of interrupt service routine is:

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

As you see, the Interrupt Service Routine is just a function pointer having one *void* type argument pointer. The client device driver or other application registers its interrupt service routine by using gic_register_int_handler() function with the parameter of interrupt number, function pointer to its service routine, and pointer to its argument. The gic_register_int_handler() function looks like below. It just registers function pointer and argument pointer to a handler vector array.

```
void gic_register_int_handler(int vec, int_handler func, void *arg)
{
        handler[vec].func = func;
        handler[vec].arg = arg;
}
```

SLOS for process management needs only one interrupt – timer interrupt and the interrupt handler vector has only one ISR for timer interrupt for now. Later, we will two more interrupt service routine for PL subsystem interrupts.

GIC interrupt handler implementation is in function gic_irq_handler(). This function implements the step 7) interrupt handler described in chapter 4.3.3.

```
uint32_t gic_irq_handler(void)
{
      uint32_t ret = 0;
      uint32_t num, val;

      /* ack the interrupt */
      val = readl(GIC_ICCIAR);
      /* cpuid is not used */
      /*cpuid = val & 0x1C00;*/
      num = val & 0x3FF;

      if (num >= NUM_IRQS) {
            return 1;
      }
      /*ret = handler[num].func(frame);*/
      ret = handler[num].func(0);
      /* clear int status bit */
      writel(1, PRIV_TMR_INTSTAT);

20          writel(val, GIC_ICCEOIR);
21          return ret;
22      }
```

In line 7, it reads the ICCIAR register. The return value of reading this register is composed of CPU ID bit[12:10] and interrupt ID bit[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 ISR from interrupt service handler vector. This interrupt ID is used for the index of interrupt service handler vector. After running the interrupt service routine, it clears the private timer interrupt status bit and write the value of GIC_ICCIAR register which is read in line 7 to ICCEOIR register. The interrupt handling steps 1) to 4) in chapter 4.4.3 are handled by the GIC hardware and we don’t care about it. We just need to properly set the GIC registers in gic_init().

4.4.5 PL to PS Interrupt

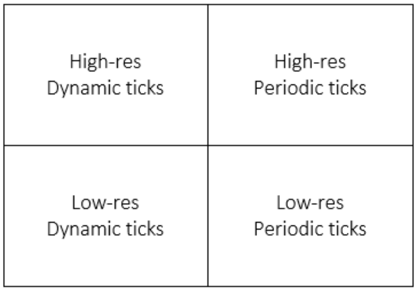
Zynq7000 has 16 PL to PS shared peripheral interrupts (SPI). Interrupt ID from #61 ~ #68, $84 ~ #91 are allocated for PL subsystem interrupts. These interrupts are useful to design custom hardware in PL subsystem and allow the peripheral hardware to signal its event to the PS subsystem. For this, we need to make a physical signal connection from a hardware in PL subsystem to GIC in the PS subsystem. We will design a simple custom coprocessor hardware and outstream device, and connect the interrupt signals from PL to PS in chapter 7 hardware-software codesign.

4.5 Timer Framework

Timer framework provides a heart beat to the process manager of SLOS. It keeps track of the time information of each task and gives this information to the scheduler. Timer framework basically provides 10 msec periodic timer tick to *Complete Fair (CFS) Scheduler* or provides other periodic timer tick to *Real Time (RT) Scheduler*. SLOS has a simple timer framework, but high-level operating system such as Linux has a well-defined timer framework. For example, following type of timer configurations in figure 4-9 are possible in Linux.

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 on, CPU wakes up by this periodic timer interrupt and wastes its battery power. Power is very important in modern mobile embedded system and this type of power loss is unacceptable.

So, recent highly matured operating system uses dynamic timer interrupt and turns off the periodic timer interrupt when the CPU goes to the low power mode.



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

Figure 4-9: Linux timer types

SLOS timer framework doesn't have such fancy and complex features. It has 10 msec periodic timer of *sched_tick* shared by all CFS tasks and another periodic timer list for realtime tasks. It maintains timer tree that has the time information of all CFS timer and realtime tasks. The timer information in this tree is used by the scheduler to pick up the next task. Before going further implementation in the timer framework, like Helloworld SLOS, we are going to implement 1 sec timer interrupt service routine (ISR) and that timer interrupt service routine will print "timer irq" message whenever it is entered.

4.5.1 Timer Interrupt Service Routine Implementation

Zynq7000 has two private timer interrupts in its PPIs (Private Peripheral Interrupt). Table 4-1 shows *Global Timer (interrupt ID #27)* for the access of all Cortex-A9 processors and *CPU private timers (interrupt ID #29)* for each core processor. SLOS timer framework uses the private timer interrupt to keep the elapsed time values. Zynq-7000 private timer hardware has following features.

1) Clock frequency is at 1/2 of the CPU frequency
2) 32-bit counter that generates an interrupt when it reaches to zero
3) 8-bit prescaler to enable better control of the interrupt period
4) Configurable single-shot or auto-reload modes
5) Configurable starting values for the counter

This is not actually used in implementing the timer framework, but as an operating system developement engineer, it is always good to know the hardware data specification.

The CPU clock frequency information can be found in *libxil/include/xparameters.h* header file. The CPU clock frequency is defined in xparameters.h as

#define XPAR_CPU_CORTEXA9_0_CPU_CLK_FREQ_HZ 666666687

This value is automatically generated by Xilinx Vivado IDE. The xparameters.h file has all hardware configuration values which is set in the Vivado IDE. Since we don't do any customization to the default PS subsystem design in chapter 2.3.7, the register information in xparameters.h must be same as the zynq-7000 technical reference manual.

Zynq7000 supports different CPU frequencies in the same part number. For example, XC7Z020 device that is used in the Zynq7000 evaluation board has 667MHz, 766MHz and 866MHz. This CPU frequency is also related to CPU power consumption. The power consumed in the CPU processor is proportional to the processor's clock frequency. So, the timer clock frequency is defined as half of the CPU clock frequency, half of CPU clock frequency is represented as 1 second time duration.

There are 4 registers related to the private timer.

1) *Private Timer Load Register*
   Register address is 0xF8F0_0600. This register contains the value copied to *Timer Counter Register* when it decrements down to zero with auto reload mode enabled.

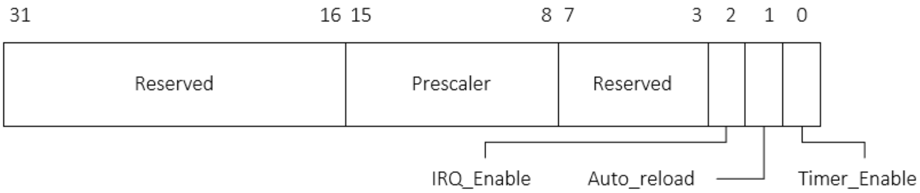
2) *Private Timer Counter Register*
   Register address is 0xF8F0_0604. This register decrements if the timer is enabled in the *Timer Control Register.* If auto reload mode is not enabled, the Timer Counter Register decrements down to zero and stops. If auto reload mode is enabled, when the timer counter value decrements down to zero, it reloads the value in the Timer Load Register and decrements from that value. When this register reaches zero, the timer interrupt status event flag is set, and the interrupt ID 29 is set as pending in the GIC's interrupt Distributor*,* if interrupt generation is enabled in the Timer Control Register*.*

3) *Private Timer Control Register*
   Register address is 0xF8F0_0608. The bits of *Private Timer Control Register* are explained in figure 4-10.
   a) Prescaler
      This value modifies the clock period for the decrementing event for the Counter Register.
   b) 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 *Private Timer Status Register*.

   c) Auto_reload
      If it is 1'b0, timer interrupt is working as single shot mode, which is if counter reaches zero, 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 zero, it is reloaded with the value contained in the Timer Load Register.

d) *Timer_Enable*
If it is 1’b0, timer is disabled and the counter value doesn’t decrement. If it is 1’b1, timer is enabled and the counter value decrements.



Figre 4-10: Bits in Private Timer Control Register

4) *Priavate Timer Interrupt Status Register*
Register address is 0xF8F0_060C. This is a banked register for all Cortex-A9 processors present. The event flag is a sticky bit that is automatically set when the Timer Counter Register reaches zero. If the timer interrupt is enabled, Interrupt ID 29 is set as pending in the interrupt Distributor after the event flag is set.

SLOS uses auto reload mode timer. SLOS dynamically programs the value in the Timer Load Register based on the information of the timer framework and updates the time quantumn value of all tasks in the timer framework as well. In this chapter, we focus on programming 10 msec periodic timer in private timer and implementing the timer interrupt service routine (ISR). After finishing the GIC hardware initialization in previous chapter, the kernel main function initializes the timer in timer_init() function. The timer_init() writes time duration value into the timer load register. The time duration is obtained through the timer rbtree. Then, enable the timer ISR in the GIC’s ISR vector table and enable the timer interrupt in the GIC. After all of these are done including other module’s initialization, the kernel main function calls the timer_enable() to trigger the timer to run. The steps to initialize timer is depicted in figure 4-11.

If timer interrupt in the GIC and timer hardware are properly enabled, the interrupt of private timer is periodically signaled to the CPU0. Then, the SLOS’s IRQ handler routine of chapter 4.3 is supposed to be executed. The GIC interrupt handler which is described in chapter 4.4.4 figures out the type of current interrupt by reading the interrupt ACK register (ICCIAR Register). As already mentioned, the return value of this register is the interrupt ID that is currently in Pending state. In this case, the return value must be the timer interrupt ID

#29. Then the timer interrupt service routine is called from the GIC ISR vector table. Since the vector table entry is the same as the interrupt ID number, the assoicated ISR routine can be called by using this number. The interrupt ID #29 is used as an index of the GIC ISR table.

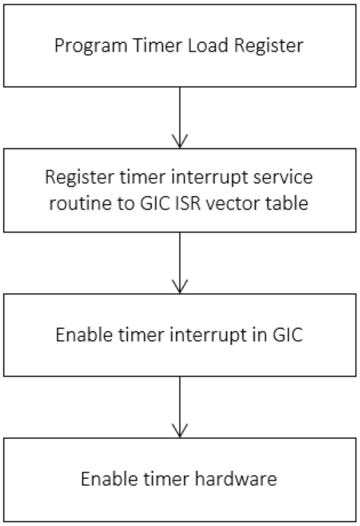

Program Timer Load Register
Register timer interrupt service routine to GIC ISR vector table
Enable timer interrupt in GIC
Enable timer hardware

Figure 4-11: Timer initialization flow

In this chapter, the timer interrupt occurs every 1 second and interrupt service routine just prints "timer_irq" message to the console. We will complete the timer service routine with the 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 infinite loop in the cpu_idle() function. The cpu_idle() routine is interrupted by timer interrupt every 1 second.

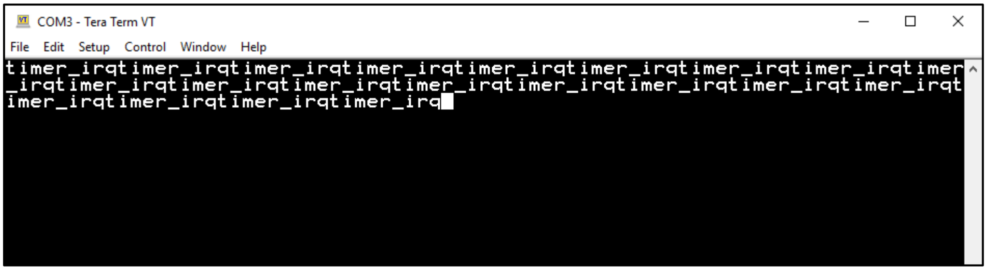
Check out the tag labed as '*SLOS_CH4_1*' by running *git checkout SLOS_CH4_1* command. This tag has a snapshot for the 1sec timer interrupt ISR. The *kernel/core* directory has only three source files - main.c, gic.c, timer.c. This tag has a simple GIC interrupt handler implementation in gic.c and timer ISR implementation in timer.c. Open the timer.c file and look for the timer_irq (void *arg) function. It just prints a "timer_irq" message to the terminal window. Build the sources and create BOOT.BIN as described before. Copy the BOOT.BIN into SD card and boot up the Zynq7000 evaluation board. See the teraterm terminal window whether there is a "timer_irq" message shown in every 1 second like figure 4-12. All these features for printing 1sec timer interrupt can be accomplished through previous chapters. This has only printing message feature in the timer interrupt service routine on top of those previous implementation.

Next, let's go through the SLOS's interrupt handling sequence step by step with our debugger tools, GDB and XSDB. Connect XSDB and GDB as described in chapter 3.8.1. After the reset of processor with *rst* command in XSDB debugger, when Zynq7000 is reset, set the break point at 0x0010_0018 which is the IRQ exception vector address. This address is the start location of interrupt handling sequence. Remember that when you set the break point, you should use hardware break point, using option '*hw*'. The correct command is *bps 0x100018 hw* in the XSDB debugger window. From the moment when debugger break the processor at reset vector, all debugging commands can be run in the GDB window. Release the processor by entering *c* command in the GDB window. Then after a couple of seconds, the break point at IRQ vector address (0x0010_0018) must be hit. Then follow the sequence by using *si* command. Running the *disassem* command can display the assembler code. With the *si* and *disassem* commands, we can perform the assembler level debugging. Then run *b gic_irq_handler* in GDB debugger window to add another break point to the GIC interrupt handler's start address.

In the gic_irq_handler() routine, you can check the GIC registers, CPU private timer registers by reading those registers' address. To read the data at specific memory address or at specific memory mapped registers, *p/x *address* or *x address* commands can be used. Look into the source files and how they work, especially check the interrupt sequence implementation explained in chapter 4.3.2.

We don't discuss this 1 sec timer IRQ in detail for now. A detailed implementation will be covered from next chapter. In this chapter, we need to know what is needed to handle interrupts and how to debug the interrupt sequence. On top of current 1 sec timer IRQ source tree, we can easily reduce it to 10msec and add timer framework.

Now, we have an implementation flowing up to the timer interrupt service routine, which covers the timer hardware is configured as triggering 1 sec timer interrupt, the Distributor and CPU interface in GIC are configured, IRQ handler and timer ISR implementations - just printing a message - are done. Next, we are going to add a timer framework to manage each task's timer statistics.



```
timer_irqtimer_irqtimer_irqtimer_irqtimer_irqtimer_irqtimer_irqtimer_irqtimer
_irqtimer_irqtimer_irqtimer_irqtimer_irqtimer_irqtimer_irqtimer_irqtimer_irqt
imer_irqtimer_irqtimer_irqtimer_irq
```

Figure 4-12: 1Sec timer interrupt service routine

4.5.2 Quick Review on Linux Timer Framework

Figure 4-13 is a layered analysis of Linux timer framework for your reference. On top of clock hardware, there is virtual and logical layers. This analysis is similar with what we did in chapter 1. Physical layer is self-explanatory. It is a hardware resource and it is a clock (timer) chip hardware in this case. Virtual layer is a virtualization of that hardware resources to expose its functional features to upper layer. To virtualize the timer chip hardware, there is a *clock_source_device* and *clock_event_device*. Logical layer is a logical implementation of the hardware resource that is abstracted by a virtual layer.

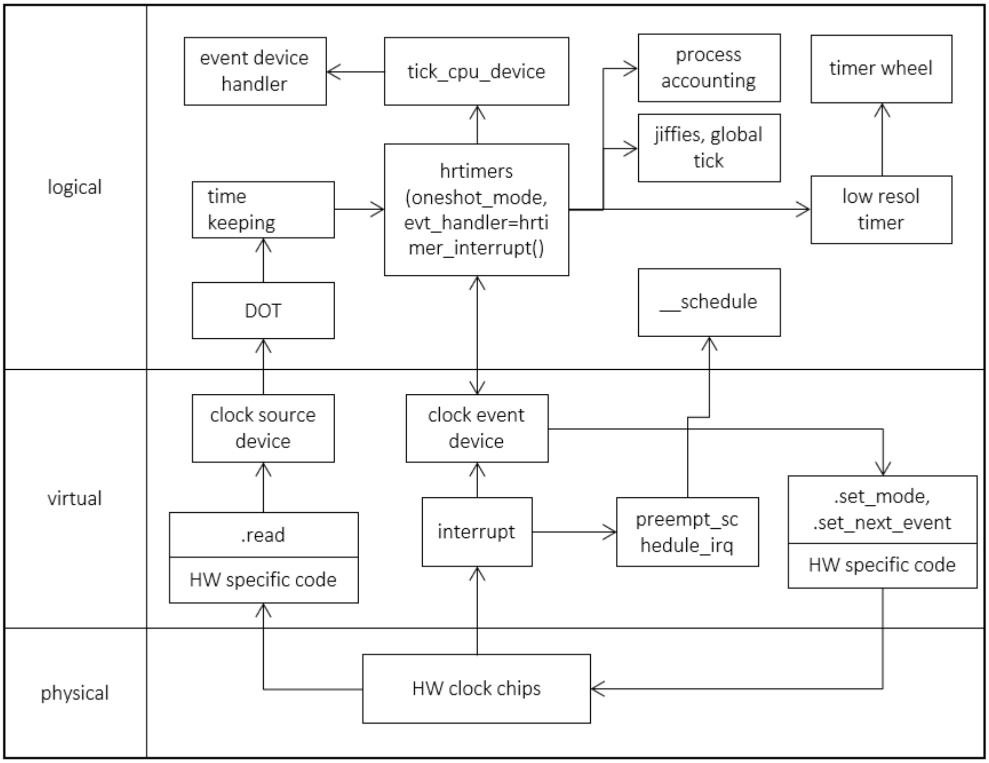


Figure 4-13: Layered model for Linux timer framework analysis

This disgram was done on Linux version 3.10 which was release long ago (if my memory is correct, it must be year 2013). As of this book is written, there is Linux version 4.19. There could be huge changes since then. Linux evolves too fast to keep up.

The reason why I brought up this diagram is that high level operating system such as Linux has a well-defined, well-designed timer framework to support diverse needs. It also virtualizes the hardware resouce very well to support different implementations for the requirements in logical layer. This diagram looks comply very well with the analysis on modern operating system in chapter 1.1.

Another reason for this diagram is that we will compare this with the similiar diagram of SLOS timer framework. SLOS's timer framework is much simpler than this, of course, but this comparison helps to understand how to virtualize hardware and add the logical implementatioins in developing the operating system. We will discuss the differences shortly after finishing SLOS's timer framework implementation.

This diagram is just for your reference. If you don't have interests in the Linux, you can dismiss this complex diagram. Moreover, this diagram could be changed now since Linux changes so fast. Look into the figure 4-13 for your reference, but let's not discuss on the Linux implementation in detail, rather let's keep track on our SLOS implementations.

4.5.3 SLOS Timer Framework Implementation

4.5.3.1 Red-Black Tree for Timer Framework

SLOS uses a *red-black* tree for maintaining its resources such as a timer tree and a *runqueue*. We should not implement the red-black tree by ourselves because it is hard to have a stable, bugless implementation. Moreover, it is a data structure and algorithm area, not an area of operating system implementation.

Thankfully, we will just borrow a good, stable and verified implementation of read-black tree from Linux source. Red-black tree is also widely used in Linux. For example, Linux uses a red-black tree to manage its runqueue. SLOS will use red-black tree in a similar way as Linux does in its runqueue management.

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

1) Red-black tree is a one of balanced binary tree
   A balanced tree guarantees the height of tree is $log\, n$, which means searching, insert, delete operation takes $O(log\, n)$ time complexity.

2) Red-black tree's insertion, removal is faster than *AVL tree*
   AVL tree is another type of self-balancing binary 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 of subtrees. With these rotations, it maintains the balance among nodes. AVL tree is reported as a better algorithm in balancing its nodes than red-black tree and this fact gives 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 occur frequently in operating system. Moreover, in operating system implementation, normally we need to pick up the next task only. This means we don't have to search the whole tree, and red-black tree in Linux keeps track of the left-most child which is, in most cases, the searching item in the tree. The left-most child could be the next scheduled task in the runqueue. By maintaining this

left-most child, the searching time in red-black tree takes $O(1)$, a constant time. So red-black tree beats the AVL tree and is selected in Linux operating system implementation.

Red-black tree is going to be used both in a timer framework and in a runqueue management of CFS scheduler in SLOS. SLOS gets red-black tree implementation from Linux and applies it to timer framework and runqueue of CFS scheduler. SLOS also keeps track of left-most child for better performance. This is pretty similar with how Linux uses the red-black tree.

4.5.3.2 New files for SLOS Timer Framework

For the implementation of timer framework, following files are added or modified on top of previous chapter (tagged as SLOS_CH4_1) under *kernel/core*. You can pull those sources by running *git checkout SLOS_CH4_3* command.

1) timer.c
   This file has a timer hardware initialization, timer hardware enable/disable routines, timer interrupt service routine (ISR) and time delay function. This file mostly includes the implementations about timer hardware.

2) 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 maintaining the elapsed time.

3) slmm.c
   This file has a *Simple & Light Memory Management (SLMM)*. Even we are working on process management, the SLOS needs to use memory like forkyi() a new task. As mentioned in chapter 4.2.3, this heap is used for application's needs. Currently, memory management has only one function of kmalloc() and it doesn't have memory free() function. We will change this file in chpater 5 with our custom memory management routines. Now, this simple heap memory management is enough for forkyi() and for timer instance allocation.

4) rbtree.c
   This file has an implementation of red-black tree. This file comes directly from Linux source tree without any changes.

4.5.3.3 Analysis on *struct timer_struct* Structure

SLOS timer framework is composed of timer instances and red-black tree. The types of timer instances are CFS timer and Realtime timer. The link between timer instance and red-black tree is done through the *rb_node*. Following is the structure of timer instance.

```
struct timer_struct {
        uint32_t tc;
        uint32_t intvl;
        timer_handler handler;
        uint32_t type;
        struct rb_node run_node;
        uint32_t idx;
        void *arg;
};
```

1) uint32_t tc
   This element is the *tick count* of remaing time until next schedule, which is the task's deadline for the scheduling. The *tc* value is decreased by the amount of elapsed time in the timer ISR. The scheduler should run the task before its *tc* value goes to zero. SLOS scheduler does its best effort to meet these deadlines of every timer.

2) uint32_t intvl
   This is the value of periodic timer. In other words, this is the time quantumn of each task. The completion time of each task must be shorter than this period duration.

3) timer_handler handler
   This is a pointer to a handler function of the timer. The timer ISR calls each timers handler function. There is only one timer handler for all CFS tasks, but each realtime tasks has its own timer handler.

4) uint32_t type
   This is a timer type which is one of *ONESHOT_TIMER* or *SCHED_TIMER* or *REALTIME_TIMER*. If the timer type is SCHED_TIMER, timer ISR calls a CFS Scheduler handler. If timer type is REALTIME_TIMER, timer ISR directly calls a context switch function.

5) struct rb_node run_node
   This is a reference to the rb_node in the red-black tree.

6) uint32_t idx
   A unique timer ID number.

7) void *arg
   A pointer to timer's argument.

Actually, each timer handler is called based on *Earliest Deadline First (EDF)* scheme. The left-most node in the red-black timer tree is the timer instance which has the least remaining time to its deadline. This means all timer instances in SLOS is a realtime timer. Sched timer itself is just another special instance of realtime timer. The time span (the *intvl* value) of sched timer is shared among all the CFS tasks. How to share the sched timer among CFS tasks will be described later along with CFS task scheduling.

4.5.3.4 Timer Framework Implementation

In this chapter, we have to focus on the timer framework implementation; how to define the timer structure, timer tree, how to create timer and add, update the timer tree.

1) How the timer tree looks like
   Below figure 4-14 is an example of timer tree. Sched timer and realtime timer are mixed in one timer tree. See the red-black tree is a 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, *ptroot*, has a pointer to the left-most node as we discussed. This left-most node is the timer's node which has the earlies deadline. Timer ISR will reprogram next timer interrupt with the tick count of the timer and maintaining the left-most node makes searching earliest timer in red-black tree take constant time.

2) Timer Handlers
   For a sched_timer, there is a unique sched_timer_handler() for managing the sched tick. To the contrary, each realtime timer has its own timer handler. The sched_timer_handler() calls the CFS Scheduler and other realtime timer handler has each realtime task's implementation. Knowing this difference is important to mix CFS tasks and realtime tasks in one timer tree. In the figure 4-14, an example of timer red-black tree, each timer, whether it is realtime timer or sched timer, has a run_node associated with its location in the red-black timer tree.

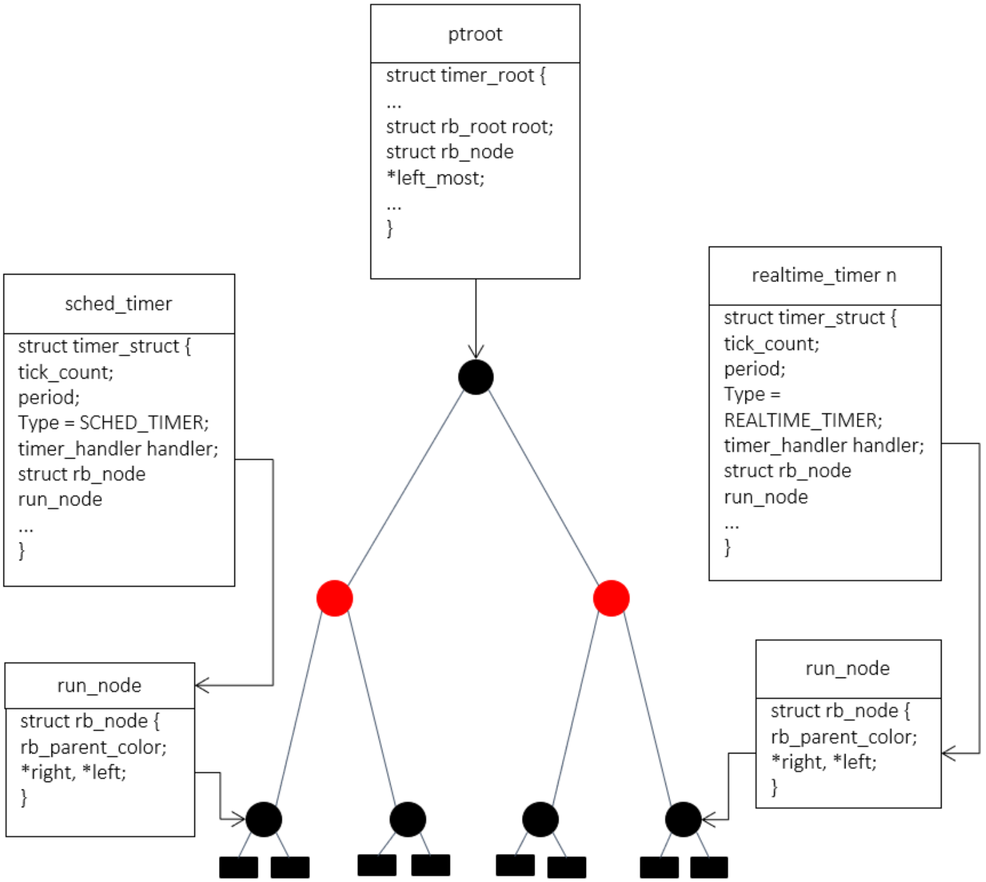


Figure 4-14: Red-Black tree implementation for SLOS timer framework

3) Initialization of timer red-black tree

The initialization of timer tree is very easy. timertree_init() function does this and is called in timer_init(). timertree_init() doesn't do anything but kmalloc() to allocate a heap for the red-black tree root, the ptroot, because there is not timer instance yet.

4) How to create a timer instance

By initializing the members of timer structure, a timer is created. There are three kinds of timer creation functions for the different timer type. create_rt_timer() is for the creation of realtime timer, create_oneshot_timer() is for oneshot timer, create_sched_timer() is for sched timer. But all of them do the same thing; take the timer handler, timer period, index and argument pointer as a parameter and initialize the timer structure and insert the timer's run_node into the red-black timer tree. The only difference is the timer's type value setting.

The realtime timer creation is done by below routines. It first allocates a heap memory for struct timer_struct in line 3, and initializes the timer structure members through line 4 ~ 10. Notice that the create_rt_timer() function takes a pointer to the realtime task in line 4 and sets realtime timer handler as NULL in line 5. The create_sched_timer() function has the task pointer as NULL because there is not a specific task for CFS tasks. Instead, the CFS scheduler is set as the timer handler function and is called in the timer_irq(). In line 12, a timer instance is added to the red-black timer tree.

```
1    void create_rt_timer(struct task_struct *rt_task, uint32_t msec, uint32_t idx, void *arg)
2    {
         struct timer_struct *rt_timer =
3                                        (struct timer_struct *)kmalloc(sizeof(struct timer_struct));
4        rt_timer->pt = rt_task;
5        rt_timer->handler = NULL;
6        rt_timer->type = REALTIME_TIMER;
7        rt_timer->tc = get_ticks_per_sec() / 1000 * msec;
8        rt_timer->intvl = rt_timer->tc;
9        rt_timer->idx = idx;
10       rt_timer->arg = arg;
11
12       insert_timer(ptroot, rt_timer);
13   }
```

5) Insertion, Deletion of timer instance

Insertion and deletion of timer instance to the red-balck timer tree is solely about red-black tree manipulation. Since we borrow this from Linux, we can refer Linux documentation [5]. It explains how to insert or delete an rb_node from red-black tree. As the document says, we have 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.

```
void insert_timer(struct timer_root *ptr, struct timer_struct *pts)
{
    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;
                struct timer_struct *entry=rb_entry(parent, struct timer_struct,
10              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  }
```

This function gets parameters of the red-black tree root and a timer that needs to be inserted. It starts from root rb_node and goes to the bottom on the tree by comparing the tick count of the timer to the tick count of current node. Remember the tick count (tc) is the remaining deadline of the timer. If the tick count of inserting timer is lesser, then the timer has an earlier deadline and it must go to the left of the current node. This is the comparison routine from line 8 to 17. Line 19~20 is for keeping the left most node. If the inserting timer has the least value and thus it is the left most timer, update the rb_leftmost of pt_root to point it. Line 22~23 is an operation for rebalancing the red-black 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.

6) Timer Interrupt Service Routine (Timer ISR)

In the previous chapter, we implemented a simple timer_irq() function. That timer_irq() function just prints a "timer_irq" message to verify there is a timer interrupt and the control flow reaching the timer ISR. Now, we have to enhance this simple timer ISR by adding the timer framework.

Below is the complete timer ISR handler. Since this timer ISR is an essential part of SLOS process management, let's look into it in more detail as below.

```
int timer_irq (void *arg)
{
    struct timer_struct *pnt, *pct;
    uint32_t elapsed;

    elapsed = get_elapsedtime();
    pct = container_of(ptroot->rb_leftmost, struct timer_struct, run_node);
    update_timer_tree(elapsed);
    pnt = container_of(ptroot->rb_leftmost, struct timer_struct, run_node);
    tc = pnt->tc;
    /* reprogram next earliest deadline timer intr */
    writel(tc, PRIV_TMR_LD);

    update_current(elapsed);

    switch(pct->type) {
        case SCHED_TIMER:
            if (current->type != RT_TASK) {
            sched_timer->handler(elapsed);
            }
            break;
        case REALTIME_TIMER:
        case ONESHOT_TIMER:
            if (current != pct->pt) {
                        switch_context(current, pct->pt);
                        pct->pt->yield_task = current;
                        current = pct->pt;
            }
            if (pct->type == ONESHOT_TIMER) {
                        del_timer(ptroot, pct);
            }
32                break;
33            default:
34                break;
35        }
36        return 0;
37    }
```

Line 6 and 8 gets the elapsed time between current timer interrupt and the previous timer interrupt, then update the timer tree. Line 9 gets next timer, and line 12 reprogram the timer hardware with the time interval of the next timer.

The timer ISR first gets the left-most rb_node from timer tree in line 7 and runs the timer handler of current timer interrupt from line 16. From line 16, the ISR calls the current timer's handler function. If current timer is CFS timer, the handler must be the CFS scheduler routine. If current timer is realtime timer, it must call the associated realtime task's handler. This sequence is described as below figure 4-15. In figure 4-15, the timer ISR gets the elapsed time by reading the private timer load register. It contains the elapsed counter value between timer interrupts. 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 tasks' *tc* value with the elapsed time value. While updating the timer tree, the deadline of current timer is reset to its original timer interval. After updating the timer tree, the left-most child is the next urgent timer and must be served before its deadline has passed. Thus, ISR always reprogram the timer hardware with the value of the left-most child of timer tree.

See the difference between the figure 4-15 for SLOS timer framework and the figure 4-13 for Linux timer framework. The SLOS timer framework, as the name implies, is much simpler. It doesn't have a virtual layer because it doesn't need to support multiple timer hardwares. It doesn't have any virtualization of the system hardware resources like clock_event_device, clock_source_device. In addition, SLOS timer framework has only the logical implementations.

4.5.4 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. SLOS sched timer has 10msec interval, but we magnify the timer intervals for this test. When the timer's handler is called, each timer prints the

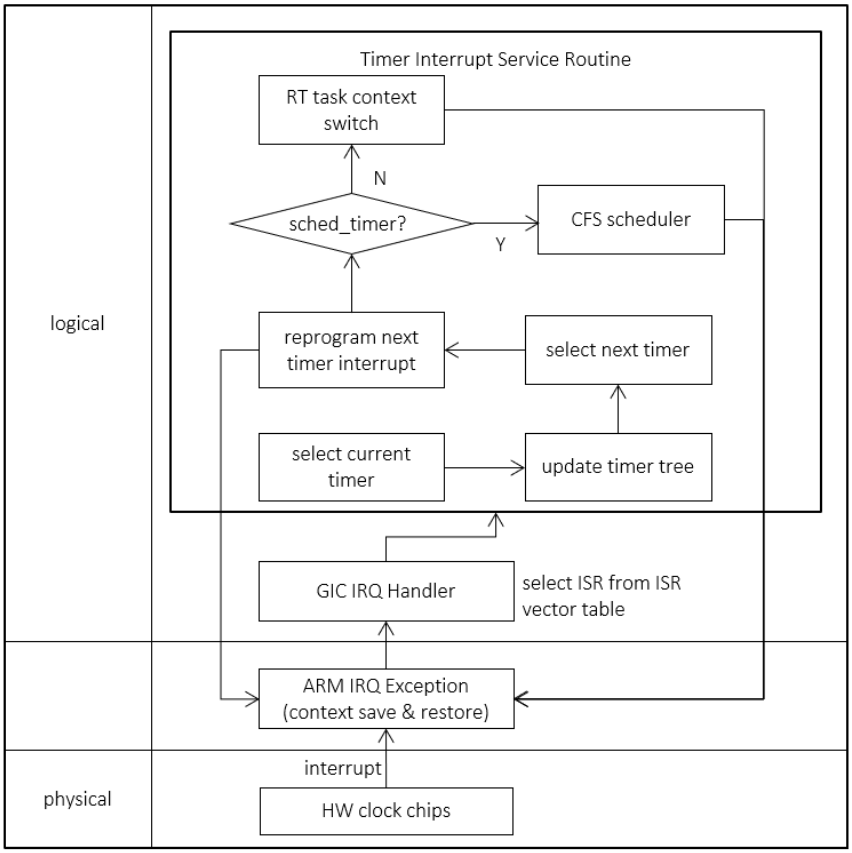


Figure 4-15: SLOS timer interrupt service routine analysis

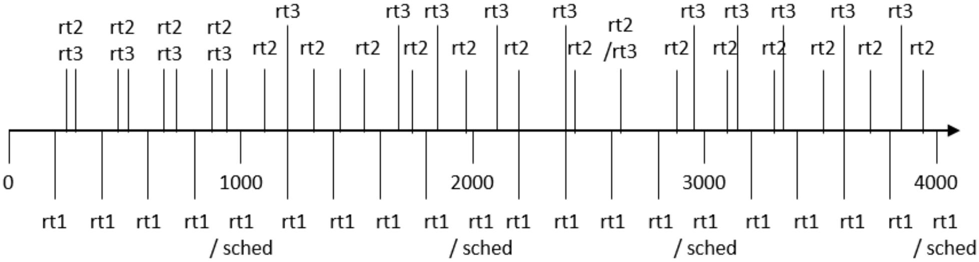


Figure 4-16: Test timers' sequence prediction

timer's name. We can predict the timer ISR should run below sequence in figure 4-16.

To run the test, check out the SLOS sources with *'SLOS_CH4_3'* tag. In order to create those 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 normal scheduler tick but it is ok for now. Currently, all the things that the sched_timer handler does is printing a message. After we implements the CFS scheduler in chapter 4.7, it will schedule the CFS tasks. create_rt_timers() creates test realtime timers. All these creation must be done after timertree_init() since adding timer means adding timer's rb_node into timer tree. The realtime timer handlers also print a message of rt_timer name into terminal with xil_print(). Build and create BOOT.BIN as we did before. After creating the BOOT.bin image, copy it to SD card and try to boot the board. You should see the message with the sequence depicted in figure 4-16.

4.6 SLOS Process Management

SLOS process management is running on top of timer framework. We have a good timer framework in previous chapter. The process management is simply either running a realtime task which is linked with the timer or running a CFS task which is scheduled by the CFS scheduler. In the figure 4-15, the timer ISR determines this. If current timer is from sched_timer, it calls CFS scheduler. If current timer is from realtime timer, it calls a context switch with the realtime task having the earliest deadline. The top-level scenario works in this way, but we have to define or to implement something more to run that scenario. This chapter will cover the management of the tasks associated with those timers.

4.6.1 Task Control Block

Task Control Block (*TCB*) is a data structure containing all information of a task. This TCB is defined as *task_struct* structure and is everything about a task. A TCB has a task's context which is a virtualization layer of a processor. TCB structure also includes task entry point, process id, linked list node for next/prev task, and so on. Below is the definition of a TCB structure. Let's look into more detail on the TCB's members.

```
struct task_struct {
    struct task_context_struct ct;
    task_entry entry;
    void *arg;
    char name[32];
    uint32_t pid;
    struct sched_entity se;
    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    };
```

1) struct task_context_struct ct
   The line2 is a *task context* member variable. A task context is a snapshot of the processor's running state. It contains the values of the ARM core registers described in chapter 2.5. Task_context looks like as below.

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

   These registers are the snapshot of a processor state while the processor is running a specific task. If a task keeps the snapshot of processor states, it can restore the processor state whenever that task comes back to the processor. These registers are stored and restored during the *context switch*. The order of these member registers is predefined and can't be changed for the correct operation of context switch.

2) task_entry entry
   This is the entry point of a specific task body. It is a pointer to a function doing a meaningful work. For now, most of SLOS's tasks are just a meaningless infinite loop except the shell task. Shell task gets user commands from console terminal through UART input and performs a corresponding work such as showing task statistics to the terminal.

3) void *arg
   This is a pointer to a task argument, but not used in SLOS. Deprecated.
4) char name[32]
   This is a character string for the task's name. Task name must be less than 32 bytes

including '\0' character.

5) uint32_t pid
This is a task's process ID. This value is assigned in task creation order.

6) struct sched_entity se
This member is used only for CFS Scheduler. struct sched_entity has all statistics values for the CFS Scheduler. What it is and how it is used in the CFS scheduler will be explained in chapter 4.7.

7) struct list_head task
struct list_head is a structure used for bidirectional linked list of tasks. All tasks are connected with each other with the struct list_head task variable. SLOS gets the task's start address by using below macro.

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

The offsetof() is a macro in *libc* library that gets an offset of a member of a struct type. The ptr is a pointer to the member of current struct instance. With the struct list_head task, we can traverse all the tasks in the task list.

8) struct task_struct *yield_task
This is a pointer to a preempted task by a realtime 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 core processor to the preempted task. This is described in chapter 4.8 in more details.

9) struct list_head waitlist
This is for the task list in a wait queue. SLOS has a *waitqueue* to queue up the tasks which is in *TASK_WAITING* state. The waitqueue will be explained in the task state discussion in chapter 4.6.3.

10) TASKTYPE type
This is a member for setting the type of a task. The type can be either CFS_TASK or RT_TASK or ONESHOT_TASK. What each type value means is quite straight forward. The timer ISR will call correct handler based on this information.

11) uint32_t missed_cnt
This is used for counting the number of missed dead line of the realtime tasks. When

updating the timer tree, it should check if there is any task missing the deadline and put it to the left-most node of the timer tree. A shell command '*taskstat'* shows the missed dead line numbers of each realtime tasks.

12) uint32_t state
This represents the task state. Task state is one of TASK_RUNNING, TASK_WAITING, TASK_STOP_RUNNING, TASK_STOP*,* but all of these states are not used. This is explained in chapter 4.6.3 along with waitqueue.

struct task_struct can have more information such as memory map list, open file list and so on. But we keep the TCB in SLOS as simple as possible. The most important thing of TCB is a struct task_context_struct structure that virtualizes the processor and is the content of the context switch.

As chapter 1 describes that operating system virtualizes the hardware resources, the TCB allows the processor to be shared with other applications also. We will develop a separate user application later which is compiled independently. Then, we will load this user application and make this program share the processor with kernel tasks through the TCB virtualization of the processor.

4.6.2 Task Creation

Before running any tasks and schedule them, we must create those tasks first. SLOS has a forkyi() function for creating a new task. A forkyi() function creates another task and enqueues it into a runqueue, then a scheduler makes the created task run. The forkyi() function is simple. It is copied below. All it does is to allocate a heap memory for a new task and initialize the content of the task's TCB. In below forkyi() function code, line 3 is for heap memory allocation of the new task. SLOS has a 64KB stack in total for all kernel tasks. Since each task's stack size is 4KB, the maximum task number in SLOS is sixteen. forkyi() function must set the stack pointer (r13) with correct stack address of each forked task. Then, it initializes the members of struct task_struct. Line 15 sets the stack start address of the newly created task. Stack area of each task is sequentially assigned in the 64KB region based in the order 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 of the task with this function address. All tasks in SLOS are running in *Supervisor mode* and line 18 does this mode-set. Line 16 and 17 are meaningless for now and are updated while task's context is switched with the other task. The rest part of line 21 ~ 25 is for maintaining the task linked list. The new task is added to the end of the task list.

```
1       struct task_struct *forkyi(char *name, task_entry fn, TASKTYPE type)
```

```
{
    pt = (struct task_struct *)kmalloc(sizeof(struct task_struct));

    strcpy(pt->name,name);
    pt->pid = pid++;
    pt->entry = fn;
    pt->type = type;
    pt->missed_cnt = 0;
    pt->se.ticks_vruntime = 0LL;
    pt->se.ticks_consumed = 0LL;
    pt->se.jiffies_vruntime = 0L;
    pt->se.jiffies_consumed = 0L;
    pt->yield_task = NULL;
    pt->ct.sp = (uint32_t)(SVC_STACK_BASE - TASK_STACK_GAP *
    ++task_created_num);
    pt->ct.lr = (uint32_t)pt->entry;
    pt->ct.pc = (uint32_t)pt->entry;
    pt->ct.spsr = SVCSPSR;

    /* get the last task from task list and add this task to the end of the task list*/
    last->task.next = &(pt->task);
    pt->task.prev = &(last->task);
    pt->task.next = &(first->task);
    first->task.prev = &(pt->task);
    last = pt;
}
```

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

```
void create_cfs_task(char *name, task_entry cfs_task, uint32_t pri)
{
    temp = forkyi(name, (task_entry)cfs_task, CFS_TASK);
    set_priority(temp, pri);
    rb_init_node(&temp->se.run_node);
    enqueue_se_to_runq(runq, &temp->se, true);
}
```

It first calls forkyi() to allocate and to initialize the new task, sets the new task's priority, and finally insert that task's *sched_entity (se)* into the *runqueue*. Runqueue for CFS task is another red-black tree in SLOS. The enqueue_se_to_runq() function inserts a sched_entity into a runqueue red-black tree. The operations in runqueue red-black tree are pretty similar with the operations in timer framework's red-black tree. The only difference is that timer red-black tree uses the deadline of tick count as node's key to be compared but red-black tree for runqueue of CFS task is using the *virtual runtime* as the key to be compared. Virtual runtime is normalized runtime with the task's priority. It goes slow in high priority task and goes faster in low priority task. The details of virtual runtime and sched entity will be explained in chapter 4.7 CFS Scheduler.

A task creation routine for realtime task is create_rt_task() function. But it doesn't need to add a new task to a runqueue and to set a priority. Currently, realtime task in SLOS doesn't have a priority. Instead, it needs to set the time interval (intvl) of the realtime task. This time interval is used for the 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 and enqueues it into the timer framework. Unlike all CFS task share one *sched timer*, all realtime tasks have their own unique timer. Realtime task creation routine looks like below. There is no sched_entity and enqueue to the runqueue routine here. Rather, it directly adds the realtime timer into the timer framework.

```
void create_rt_task(char *name, task_entry handler, uint32_t dur)
{
    temp = forkyi(name, (task_entry)handler, RT_TASK);
    temp->timeinterval = dur;
    temp->state = TASK_RUNNING;
    create_rt_timer(temp, dur, rt_timer_idx++, NULL);
}
```

Another task type in SLOS is a *oneshot task* which runs only one-time timer. Oneshot task creation is done through create_oneshot_task() function and is same as realtime task except its oneshot timer is removed before oneshot task runs. Removing oneshot timer is done simply by deleting its run_node from the timer red-black tree.

4.6.3 Task State

There are two states of task in SLOS. The state of SLOS task can be either *TASK_RUNNING* or *TASK_WAITING*. CFS task can be either TASK_RUNNING or TASK_WAITING state. Currently, the realtime task and oneshot task have TASK_RUNNING state only.

To change a task's state to TASK_WAITING state, SLOS maintains a *waitqueue*. Waitqueue is a linked list of struct list_head members of waiting tasks. struct task_struct has a list_head member (struct list_head wait_list) for task's linked list implementation. Below figure 4-17 is an example of tasks' linked list. In this list, there are 5 tasks. Task 1, task 3 are in TASK_WAITING state and waiting in waitqueue. Others are in TASK_RUNNING state.

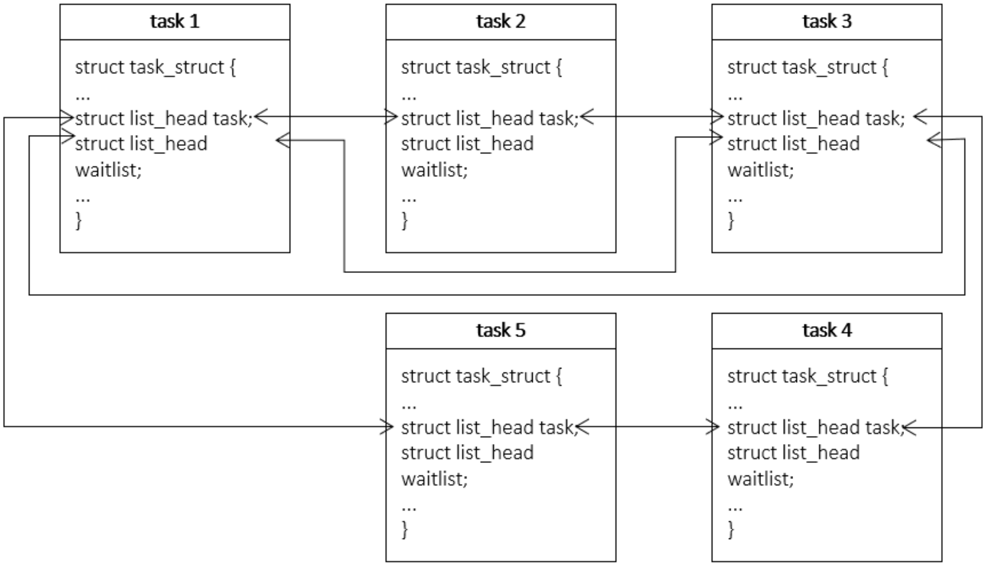


Figure 4-17: An example of task list and wait list

SLOS can calculate the address pointing to the start of a task object in memory from this list_head member by using *container_of* macro. container_of macro looks like below and is defined in *kernel/inc/defs.h*.

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

The container_of macro gets the pointer (*ptr*) to a struct list_head of a specific task, struct type name (type) and the member name (member) of that structure. container_of macro uses a offsetof() function in libc library which calculates the offset byte of a member variable from the start of struct type. This offset is also simply calculated by below macro.

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

The offsetof() macro and container_of() macro mixed with struct list_head structure is an easier way to make a new linked list. This method is widely used in Linux. SLOS also uses container_of() and offsetof() macros in the same way for its linked list of task and waitqueue. If we have a task's list_head (struct list_head task), then we can get the pointer to that specific task by following way.

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

Changing a task's state to TASK_WAITING means removing the task's sched entity from runqueue and sending that task to waitqueue 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() function changes a task's state to TASK_WAITING, and enqueue_se_to_runq() changes a tasks state to TASK_RUNNING. Those two functions follow steps in figure 4-18. For a detailed implementation, refer *kernel/core/task.c* source file.

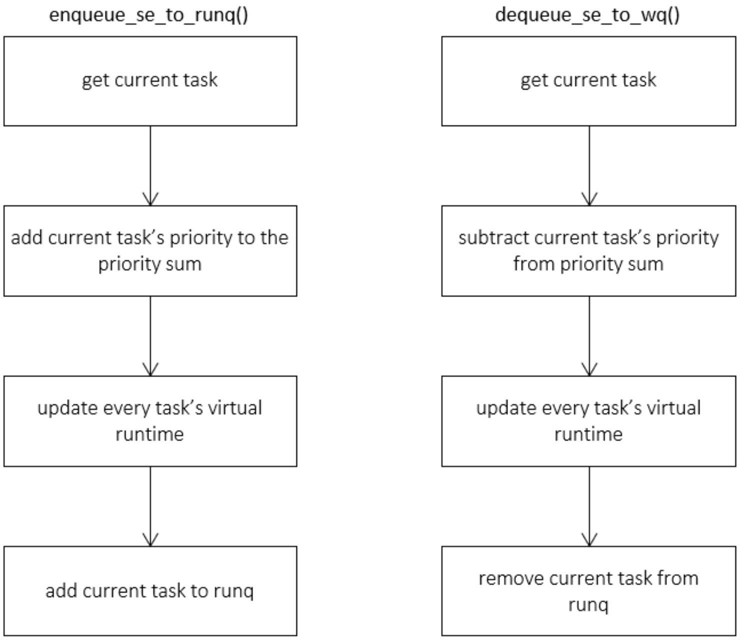

enqueue_se_to_runq()
get current task
add current task's priority to the priority sum
update every task's virtual runtime
add current task to runq
dequeue_se_to_wq()
get current task
subtract current task's priority from priority sum
update every task's virtual runtime
remove current task from runq

Figure 4-18: Enqueue, Dequeue task with runqueue and waitqueue

From SLOS v2.0, the sleep functions are supported. When a task calls msleep() or usleep(), it registers a oneshot timer for the duratioin, goes to waitqueue and changes its task state to *TASK_WAITING*. When the associated oneshot timer is expired, that task goes back to *TASK_RUNNIG* state. This task state switching will be demonstrated at the end of chapter 8.

4.6.4 Context Switching

Context switching is a core technology for multitasking in one core processor. In a commercial high-level operating system, there are hundreds of tasks waiting for running but the processor core number is much less than this client number. To share one processor core among multiple tasks, context switching between tasks is necessary. This happens so fast that the user feels multiple tasks are running simultaneously in a core. Each task stores its lastest state of running into its context variable when it is switched out. Task's running state or task's context is the snapshot of the processor's general-purpose registers. SLOS's task context storage variable is in struct task_context_struct structure. It looks like as 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 ARM processor's general-purpose registers; r0 ~ r12, sp, lr, pc, and spsr. These are already covered in chapter 2.5. Thus, switching context means saving processor registers to current task's sturct task_context_struct variable and restore next task's struct task_context_struct variable into the processor registers.

Notice that SLOS's context switching happens preemptively. While current task is running, if timer framework chooses another task's deadline is more urgent than current task, then current task is preempted by the urgent task; current task is switched out and next task is switched in while returning from timer interrupt handler. The do_context_switch(prev, next) fucntion does switching context. It is following steps in figure 4-19.

Steps in figure 4-19 looks quite similar with figure 4-6 interrupt handler. The preemptive context switching just changes the content of *context memory* from the previous task's context to the next task's context. Steps of context switching are as follows.

1) Timer interrupt fires and the interrupt IRQ handler saves current task's context into a *CONTEXT_MEM* which is located at 0x0000_4000.

2) The GIC's interrupt handler routine finds the current interrupt ID by acknowledge ICCIAR register. For now, the interrupt ID must be the CPU private timer interrupt ID which is 29. Then, it calls timer interrupt ISR function. The timer ISR will check the current timer interrupt type.

3) If current timer interrupt type is sched_timer interrupt, then it calls the CFS scheduler handler. If current timer interrupt is RT timer interrupt, then it calls do_switch_context() directly. The realtime timer has a pointer to an associated task, a direct call of do_switch_context() is possible.

4) In a CFS scheduler, the scheduler finds the worst unfair task and if that task is different with current task, then it calls do_switch_context() to switch the content of CONTEXT_MEM with worst unfair task's context.

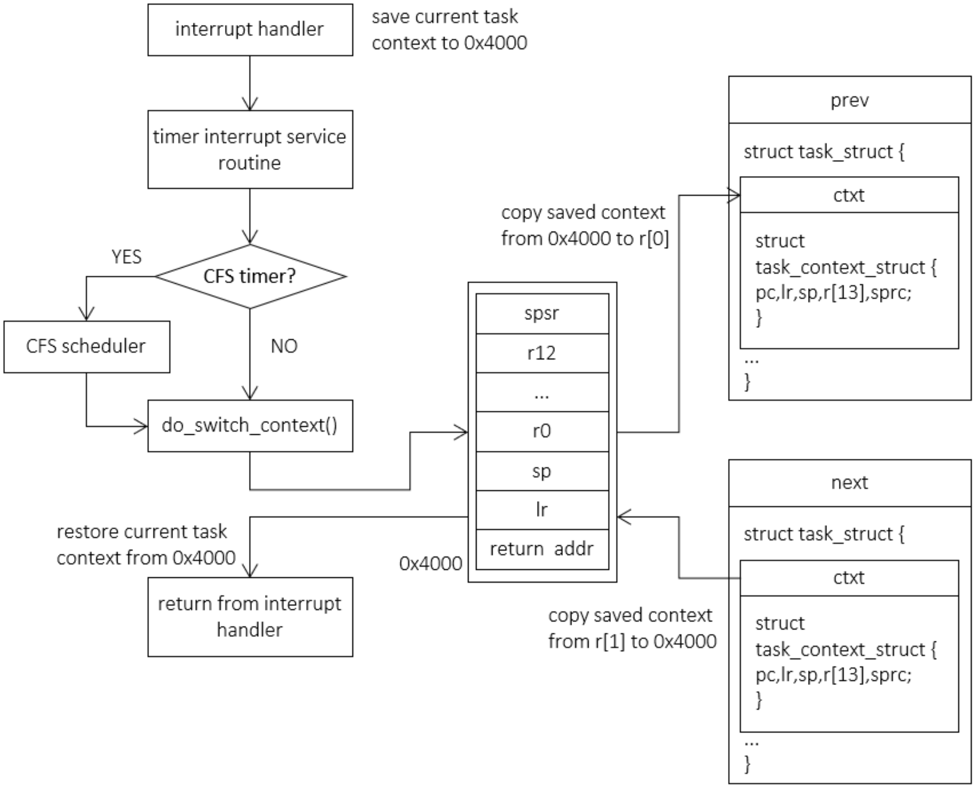

interrupt handler
save current task context to 0x4000
timer interrupt service routine
CFS timer?
YES
NO
CFS scheduler
do_switch_context()
restore current task context from 0x4000
return from interrupt handler
spsr
r12
...
r0
sp
lr
return addr
0x4000
copy saved context from 0x4000 to r[0]
copy saved context from r[1] to 0x4000
prev
struct task_struct {
ctxt
struct task_context_struct {
pc,lr,sp,r[13],sprc;
}
...
}
next
struct task_struct {
ctxt
struct task_context_struct {
pc,lr,sp,r[13],sprc;
}
...
}

Figure 4-19: Preemptive context switching in SLOS

5) do_switch_context(prev, next) routine copies the values at 0x0000_4000 to current task's context storage variables and restores the next task's context values to 0x0000_4000.
6) When returning from interrupt, the values saved at 0x0000_4000 are restored to the processor's general-purpose registers and return to the next task's return address (next task's lr register value). It is important to know that when interrupt IRQ handler returns, it returns to the next task's return address which was saved before.

The core routine of context switching is in *ops.S*. It is basically a swap-out and swap-in loop.

```
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     savectxt:
7                 ldr       r3,[r4],#4
8                 str       r3,[r0],#4
9                 subs      r2,r2,#1
10                bne       savectxt
11                mov       r2,#0x110
12                mov       r4,#CONTEXT_MEM
13    restrctxt:
14                ldr       r3,[r1],#4
15                str       r3,[r4],#4
16                subs      r2,r2,#1
17                bne       restrctxt
18                pop       {r0-r4}
19                mov       pc,lr
```

This assembler routine is composed of two parts: saving current task's context from address 0x0000_4000 (line 6 to 10) to the current task's context variable, and restoring next task's context to address 0x0000_4000 (line 14 to 17). The pointers to the current task and next task are delivered as arguments which are referred as register r0, and r1. Since this context switch routine assumes the first region pointed by the pointer is the context variable in predefined order. We shouldn't change the order struct task_context_struct variable. These routines are '*for*' loops copying 17 registers (return addr, sp, lr, r0~r12, spsr) of tasks' context.

4.6.5 Synchronization

When multiple tasks are running, there could be a contention between tasks to access a common resource. This common resource could be either a hardware resource or a software resource. Especially, the common resource that needs a serialized access is called a *critical section*. When multiple tasks try to access the critical section concurrently, there could be a racing condition between tasks. This results in a non-deterministic output of the critical section. If the output of computer isn't deterministic like human mind, how can we use it in daily life?

Operating system developes techniques to synchronize these simultaneous requests to the critical section from different tasks. Basically, a software itself including operating system can't solve the concurrency issues. Operating system depends on the hardware to achieve the synchronization of the critical section. Most processor architectures support a way of atomic access to the memory access. ARM architecture has an *LDREX (Load Exclusive)* and *STREX (Store Exclusive)* instruction for an atomic load and store process. We will use these instructions to implement a mutual exclusive access to a critical section in SLOS.

A synchronized execution in a UP (Uni-Processor) system can be easily achieved by interrupt disable/enable (This is also done with the help of hardware!). If interrupt is disabled before entering a critical section, there is no way that other tasks can preempt the current task in a single core system. After finishing the access to the critical section, the interrupt is re-enabled. Other task can be scheduled again and gain the access to the critical section. This method is the simplest way to implement task synchronization in single core system. But there are many downsides in this method [6]. First, this approach requires the operating system to allow the non-previleged task to run a previleged operation. Imagine that a non-trusted application in a system disables the interrupt and runs an infinite loop. Then, there is no way to kick off this application consuming the whole CPU time. Second, as we already mentioned, this method doesn't work in MP (Multi-Processor) system. Tasks running in other processors still can access the critical section which is already owned by some task running in different core. Third, turning off interrupts can lead to interrupts becoming lost, which can lead to serious system problems. Finally, disabling interrupt is quite inefficient, has a poor performance. All other tasks should wait until the current task finishes its job and enables the interrupt again. Even a highest priority task also has to wait the interrupt enabled in this case.

There are a couple of methods in operating system to synchronize multiple tasks' access to a critical section. *Semaphore*, *mutex* and *spinlock* are used for this purpose. SLOS has interrupt enable/disable functions (enable_interrupt, disable_interrupt), and spinlock_acquire, spinlock_release. Interrupt enable/disable functions are not used in SLOS. Rather, the spinlock, spinunlock implementations with LDREX and STREX instructions are used for synchronization of critical section. All these implementations are in *kernel/core/ops.S.* We will have a look at on the spinlock synchronization methods.

4.6.5.1 Test And Set (Automic Exchange)

Because disabling interrupts doesn't work on multiple processors, system designers started to invent a new hardware support for locking. The content of this chapter mostly comes from the reference [6].

The simplest hardware for locking is what is known as a *test-and-set* instruction, also known as *atomic exchange*. This is a base idea for spinlock implementation in SLOS. The idea is quite simple: use a simple flag (a variable in memory) to indicate whether some thread has a possession of a lock. The first thread that enters the critical section will call spinlock_acquire(), which tests whether the flag is equal to 0, and then sets the flag (a specific location in memory) to 1 to indicate that the thread now holds the lock. When finished with the critical section, the thread calls spinlock_release() and clears the flag, thus indicating that the lock is no longer held. If another thread happens to call spinlock_acquire() while that first thread is in the critical section, it will simply spin-wait in the loop for the previous thread to release and clear the flag. Once that first thread does so, the waiting thread will fall out of the loop, set the flag to 1 for itself, and proceed into the critical section.

Unfortunately, this method has two problems: one of correctness, and another of performance. The correctness problem happens if there is not an atomic exchange. This is simple to see once you get used to thinking about concurrent programming. Imagine the code interleaving between thread A and thread B; assume flag=0 to begin. As you can see from this interleaving, we can easily produce a case where both threads set the flag to 1 and both threads are thus able to enter the critical section. For example, thread A check the flag and gets the result is flag = 0. But just right before the thread A sets the flag = 1, a timer interrupt fires and the scheduler kicks out the thread A and runs the thread B. Thread B preempt thread A. In this case, thread B still gets the flag = 0 and enters the critical section as well. This behavior is what professionals call 'bad' – we have obviously failed to provide the most basic requirement: providing *mutual exclusion*.

The performance problem is the fact that the way a thread waits to acquire a lock that is already held: it endlessly checks the value of flag, a technique known as spin-waiting. Spin-waiting wastes time waiting for another thread to release a lock. The waste is exceptionally high on a uni-processor, where the thread that the waiter is waiting for cannot even run (at least, until a context switch occurs)! Thus, as we move forward and develop more sophisticated solutions, we should also consider ways to avoid this kind of waste.

The spinlock implementation of SLOS solves the first problem with the help of a hardware - an ARM ISA. ARM ISA has an *atomic exchange* instructions and it is possible to do mutual exclusion in spinlock_acquire() / spinlock_release(). But the performane loss by processor's spinning to wait unlock is not considered in SLOS. SLOS could put that task into a wait queue and yield the processor to next urgent task. This could be a future implementation.

4.6.5.2 SLOS Spinlock Implementation by Test And Set

spinlock_acqure()/spinlock_release() in SLOS allows a mutual exclusive access to a critical section by using a *'Test And Set'* method. It is done with the help of hardware's ability of an atomic access to a memory. ARM has a special hardware and instructions for this. It has an *exclusive monitor* hardware [8] [10], and ARM ISA has instructions to support *exclusive* *load/store* to a memory address. An exclusive monitor is a simple state machine, with the possible states *open* and *exclusive*. For this, there are two types of exclusive monitor; local exclusive monitor per core processor and global exclusive monitor for multicore system. To support synchronization between processors, a system must implement both monitors, *local* and *global*. Figure 4-20 shows a local exclusive monitor. Global exclusive monitor uses the core number for its state machine transition. We will look into local exclusive monitor only. For more information on global exclusive monitor, refer [10].

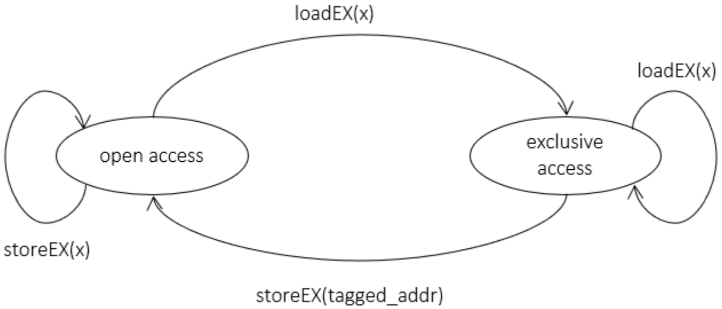


Figure 4-20: Local exclusive monitor state machine diagram

The state machine in figure 4-20 looks simple. It has two states (open, exclusive) and LDREX makes the exclusive access state and STREX makes the open access state. The *tagged_addr* in the state machine means the address where the LDREX is called. Let's look into these transitions in detail.

1) *Exclusive Monitor*

   A Load-Exclusive *(LDREX)* operation updates the monitors to exclusive state. A Store-Exclusive *(STREX)* operation accesses the monitor(s) to determine whether it can complete successfully. A STREX can succeed only if all accessed exclusive monitors are in the exclusive state. STREX instruction makes the state transition to open state whether it succeed or not. 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.

2) *LDREX*

   The *LDREX* instruction loads a word from memory, tagged the address of LDREX instruction into the monitor (*tagged address)*, 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) '*exclusive*' state and the address of LDREX instruction is tagged.

3) *STREX*

   STREX works like a conditional store command. If it succeeds in storing data to a memory location, it returns the status value 0 to a register. If it fails, it returns the status value 1 to a register. The result of a STREX operation depends on the state machine and tagged address for the processor issuing the STREX instruction. 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*. Whether it is successful or not, STREX instruction makes the state machine go back to open state. So, if the operation is failure, the atomic operation should start from LDREX again.

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

```
spin_lock_acquire:
        ldr         r1,=LOCKED
loop1:
        ldrex       r2,[r0]
        cmp         r2,r1
        beq         loop1
        /* store r1 to [r0], r2 is result */
        strexne     r2,r1,[r0]
        cmpne       r2,#1
        beq         loop1
        /*lock acquired*/
        DMB
        bx          lr
.global spin_lock_release
spin_lock_release:
        ldr         r1,=UNLOCKED
        DMB
        str         r1,[r0]
        bx          lr
```

spin_lock_acquire() first loads exclusively the value in memory address of first argument, [r0] (line 4). For *test*, it compares the result with the value LOCKED(0x1). If they are equal (flag is already *set*), it means other process already writes LOCKED value to this address (r[0]) and held the lock. Thus, current process has to spin until the UNLOCKED is written to address r[0] by the other process. This is a ‘Test’ step of a ’Test And Set‘ process. If the flag is unheld, current process tries to store LOCKED value exclusively to ddress r[0] (line 8). If it fails, current process goes back to LDREX and starts to spin again until it successfully writes LOCKED value to the address r[0]. This is a ’Set’ step. spin_lock_acquire() follows below steps.

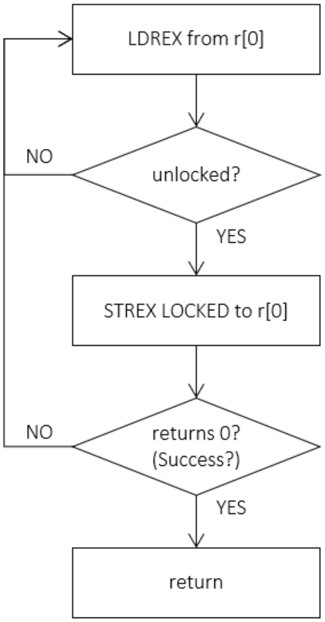


Figure 4-21: Spin_lock_acquire() by *Test And Set* method

spin_lock_release() (line 15 ~ 19) in SLOS is quite straight-forward, simply writing UNLOCKED (0x0) value to the memory address r[0].

To use the spinlock properly in multiple processors, the address r[0] for the flag should not be placed in the specific processor’s local cache. So, the spin_lock flag should be declared as a *volatile* keyword. This is an example of how to use a spinlock in SLOS.

```
volatile int rq_lock;
void funcA() {
spin_lock_acquire(&rq_lock);
/* do something useful */
5        spin_lock_release(&rq_lock);
6        }
```

Notice that the line 1 declares the spinlock flag variable rq_lock as a volatile variable. This keyword makes the flag variable in the outer memory. The spin_lock_acquire() / spin_lock_release() will do the mutual exclusive access to this memory (flag variable) with the help of hardware. But when a task holds this spinlock flag, other tasks should be spining until it is kicked out by the preemptive scheduler, or until the task holding the spinlock flag releases the spinlock.

4.6.5.3 Synchronization mechanisms in the Linux kernel

Now, we have our implementation of synchronization method, spinlock. It is worth looking into the synchronization methods widely used in modern operating system, Linux. The descriptions in this chpater mostly comes from reference [9].

When porting software from a single core environment to run on multi-core cluster, there can be situations where you must modify code to enforce a particular order of execution or to control parallel access to shared peripherals or global data. The Linux kernel (like other operating systems) provides a number of different synchronization primitives for this purpose. Most such primitives are implemented using the same architectural features as application-level threading libraries like Pthreads. Understanding which of these is best suited for a particular case will give software performance benefits. Serialization and multiple threads contending for a resource can cause suboptimal use of the increased processing throughput provided by the multiple cores. In all cases, minimizing the size of the critical section provides best performance.

1) Completions

   Completions are a feature provided by the Linux kernel that can be used to serialize task execution. They provide a lightweight mechanism with limited overhead that essentially provides a flag to signal completion of an event between two tasks. The task that is waiting can sleep until it receives the signal, using wait_for_completion (struct completion *comp) and the task that is sending the signal typically uses either complete (struct completion *comp), that will wake up one waiting process, or complete_all (struct completion *comp) that wakes all processes that are waiting for the event. Kernel version 2.6.11 added support for completions that can time out and for interruptible completions.

2) Spinlocks

   A spinlock provides a simple binary locking mechanism, designed for protection of critical sections. It implements a busy-wait loop. A spinlock is a generic synchronization primitive that can be accessed by any number of threads. More than one thread might be spinning for obtaining the lock. However, only one thread can obtain the lock. The waiting task executes spin_lock (spinlock_t *lock) and the signaling task uses spin_unlock(spinlock_t *lock). Spinlocks do not sleep and disable pre-emption.

3) Semaphores
   Semaphores are a widely used method to control access to shared resources, and can also be used to achieve serialization of execution. They provide a counting locking mechanism that can cope with multiple threads attempting to lock. They are designed for protection of critical sections and are useful when there is no fixed latency requirement. However, where there is a significant amount of contention for a semaphore, performance will be reduced. The Linux kernel provides a straightforward API with functions *down(struct semaphore *sem)* and *up(struct semaphore *sem)* to lower and raise the semaphore. Unlike spinlocks, which spin in a busy wait loop, semaphores have a queue of pending tasks. When a semaphore is locked, the task yields, so that some other task can run. Semaphores can be binary (in which case they are also mutexes) or counting.

4.7 CFS Scheduler

Now, we have an interrupt controller, a good timer framework, a task control block to abstract the processor and spinlock for task synchronization. The last piece to manage processes is the module selecting a process (task) that holds the core processor. That is the scheduler's job. Different operating system has a different philosophy in scheduling its processes. So SLOS does!

SLOS has two scheduler schemes; *CFS Scheduler (Complete Fair Scheduler)* and *Realtime Scheduler*. CFS scheduler is using a sched_timer tick variable which is periodically expired. The sched_timer is also one of the realtime timers. SLOS's CFS scheduler has a 10msec realtime timer. This timer quantum is shared through all CFS tasks. Fairness in CFS scheduler is evaluated according to the priority of the task. CFS scheduler measures this fairness with each task's *virtual runtime* which is normalized with all CFS tasks' priority sum. The speed of the virtual runtime of each task is related to that task's priority. The virtual runtime of a high priority task is going slower, whereas 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 has a bigger chance of being scheduled than low priority task to balance the fairness between them.

This scheme of SLOS's CFS scheduling imitates the Linux CFS scheduler. The concept of virtual runtime, sched_entity, is very similar with Linux. In addition, the implementation in

SLOS is similar with Linux in that it is using red-black tree, run_node and linking the sched_entity with red_black tree.

4.7.1 Run Queue, Sched Entity, vruntime, jiffies

CFS scheduler measures the sched_time tick with a variable *jiffy*. The cfs_scheduler() function which is a sched_timer's handler increases the jiffy value whenever sched_timer interrupt occurs. jiffy is a time quantumn of CFS all tasks. Since CFS sched_timer has 10msec duration, each sched_timer tick or jiffy has 10msec interval.

CFS scheduler maintains each task's fairness using a *struct sched_entity* structure. The struct 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 structure stores all information necessary for the CFS scheduler. jiffies_vruntime is a virtual runtime measured with jiffy count. ticks_consumed is a timer's tick count and jiffies_consumed is a jiffy count while a task is running. ticks_consumed is a timer tick count coming from the CPU private timer counter hardware and jiffies_consumed is a jiffy count which has 10 msec resolution. run_node is associated with a node in a runqueue red-black tree. As you can see, sched_entity structure holds all information needed for CFS scheduler. Each CFS task has its own *sched_entity* as its member variable.

The measurement unit of virtual runtime is a jiffy which is increased by one at every 10msec. The jiffies consumed by current task is weighted with its priority and is normalized with the sum of all CFS tasks' priorities. The virtual runtime of a task 'A' can be calculated as below.

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

Equation 4-1: Calcuating the virtual runtime

The priority of task A becomes higher as the number of priority goes lower and this priority should not be less than or equal to zero. So, the priority value 1 is the highest priority value. Since the virtual runtime is weighted by its priority, 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 speed is exactly 2 times slower than task B's virtual_runtime speed. In this case, the CFS scheduler which is completely fair based on the virtual_runtime will run task A two times more than task B to balance the fairness between them. A 'Complete Fair' means a fairness proportional to their priority.

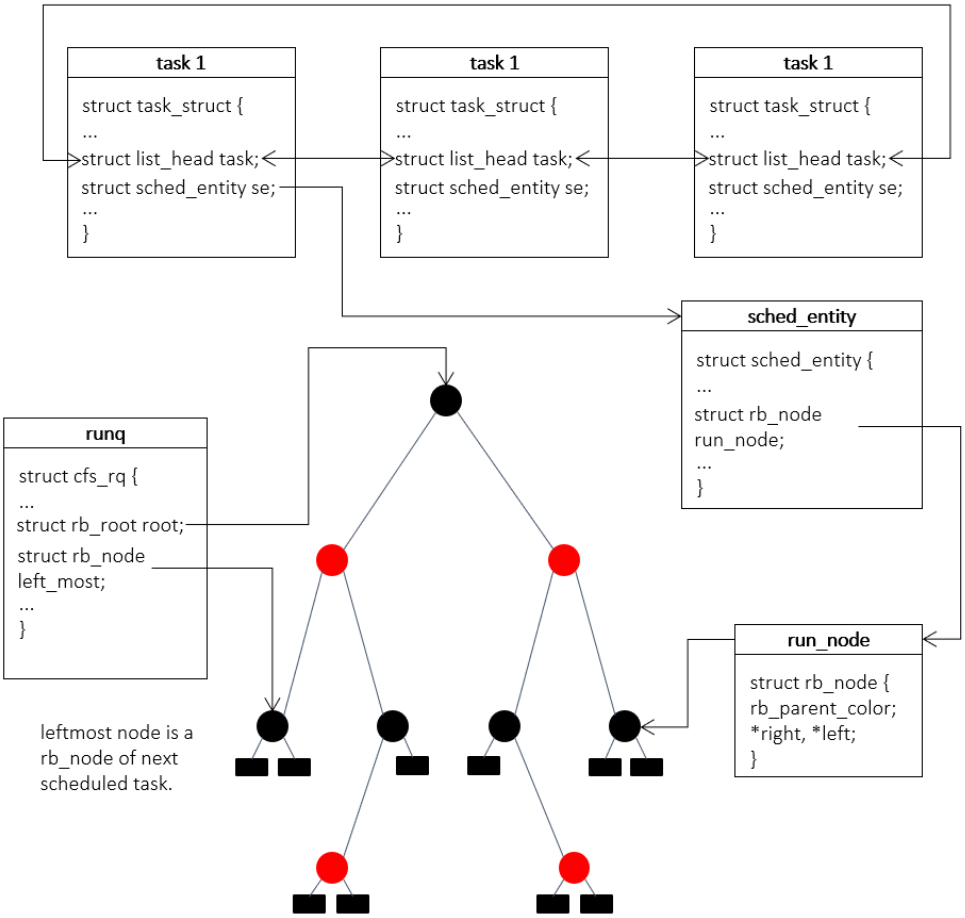


Figure 4-22: The structure of CFS tasks' runqueue red-black tree and its connection with sched_entity of CFS taskAll CFS tasks have a node in CFS runqueue. A runqueue is a red-black tree of run_node of sched_entity. CFS runqueue structure is quite simple which has a pointer to the root rb_node, a pointer to left most rb_node, and a priority sum. So, the struct cfs_rq structure 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;
};
```

A rb_node of each task's sched_entity is linked with a node in a red-black tree of runqueue. The runqueue and CFS tasks are connected like a figure 4-22. It looks pretty similar with the timer framework red-black tree in figure 4-14. Notice that CFS runqueue keeps the left-most rb_node for the task scheduled for next running on the processor.

A detailed operation of red-black tree was already covered in chapter 4.5.3.4. The operation of runqueue red-black tree is exactly same except it is using jiffies_vruntime value as its comparison key. Also, SLOS's runqueue in figure 4-22 is the same way how Linux manages its runqueue: the link among CFS tasks, sched_entity, run_node and runqueue.

4.7.2 Wait Queue and Task

SLOS has a waitqueue for storing tasks whose state is in *TASK_WAITING* state. Waitqueue is just a linked list of tasks. For this, struct task_struct has a *struct list_head waitlist*. Waitqueue is a bidirectional linked list of this waitlist variable.

As described in chpater 4.6.3, to add a task to the waitqueue, *dequeue_se_to_wq()* function is used. To move a task back to runqueue, *enqueue_se_to_runq()* is used.

4.7.3 *schedule()* Function

The CFS scheduler is called from the scheduler timer's timer handler. How to call the sched_timer handler was already covered in chapter 4.5.3.4, Timer Framework Implementation. The sched_timer handler is implemented in cfs_scheduler() function that is defined in *kernel/core/ktimer.c* and the cfs_scheduler() does only two things: first, updates the current task's sched_entity and next, it calls schedule() function. After all values including virtual runtime in sched entities have been updated, the schedule() function just calls switch_context() function when fairness is broken, i.e. current task holds the processor too much.

We can measure the fairness in a quantitative way. The measurement of fairness starts from timer interrupt service routine. Timer interrupt service routine gets the elapsed time from timer framework. This elapsed time is measured with tick count of the sched_timer. The current task's sched_entity is updated with this elapsed time tick count and the CFS runqueue is refreshed with the current task's new virtual runtime. This is done in update_se() function in *kernel/core/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 all member values of sched_entity of current task. Notice that the jiffies_vruntime is calculated in line 5 as equation 4-1. If current task is realtime task, it just updates the ticks_consumed value of realtime task as in line 8.

After updating current task's sched_entity, the sched_timer interrupt service routine, the cfs_scheduler function, calls the schedule() function that is defined in *kernel/core/task.c*. After updating the runqueue, the schedule() function can pick up the task suffering from unfairness which is the left-most node in runqueue. If that task is not the current task, schedule() calls the context_switch() which switches the content of CONTEXT_MEM with the context of the task in left-most node. This switch_context() routine is defined in *kernel/core/ops.S* and was desribed in chapter 4.6.4.

4.7.4 CPU Idle Task

The main() function of SLOS (this will be renamed as start_kernel() later), the kernel's main entry function, falls into an *cpuidle* task after all SLOS bootup processes are finished. The cpuidle task in SLOS is just a meaningless infinite loop. But normally, it works differently as what the name implies, this task has the lowest priority and should be ran only when there is not a task by the scheduler. Because current CFS scheduler in SLOS is fair to all CFS tasks, and cpuidle task is also one of CFS tasks, the cpuidle task is periodically run with the fastest virtual runtime.

In high level operating system, such as Linux again, the cpuidle task is not meaningless, rather it does a very important job regarding power management. Since the cpuidle task is scheduled whenever there is no job to run, this is a good time for making the processor go into its low power mode or sleep. Recently, the processor power management becomes very important, escpecially in mobile devices. Mobile chipset vendors spend huge amount of efforts to optimize the power consumption in the hardware-wise and software-wise as well.

There are different power mode of processor and sophisticated ways to control those processor power mode. But we are simple, we just adds a simple ARM *'WFI'(Wait For Interrupt)* instruction into a cpuidle in SLOS for demonstration. This is an example to add a smallest power saving code into SLOS but there is not a real hardware-wise implementation for this such as clock gating, power down bus and power rails to the core. We will study more on processor's power management at the end of chapter 4. Below is the new cpuidle task.

```
1     void cpuidle(void)
2     {
3         uint32_t i = 0;
4         xil_printf("I am cpuidle.....\n");
5         while (1) {
6             if (show_stat) {
7                 xil_printf("cpuidle is running....\n");
8             }
9             if (i == 0xFFFFFFF) i = 0;
10            else i++;
11            asm ("DSB" :::);
12            asm ("WFI" :::);
13        }
14    }
```

Up to line 10, cpudile() function does a dummy job. Line 11 ~ 12 is running a *'WFI'* instruction. ARMv7 reference document [3] says WFI instruction is used for :

1) Forces the suspension of execution, and of all associated bus activity
2) Suspends the execution of of instructions by the processor

The cpuidle 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.

The WFE (Wait For Event) and WFI (Wait For Interrupt) instructions enable you to stop execution and enter a low-power state. To ensure that all memory accesses prior to executing WFI or WFE have been completed (and made visible to other cores), you must insert a *DSB* instruction. Detailed descriptions on memory barrier are followed in section 5.3.4.

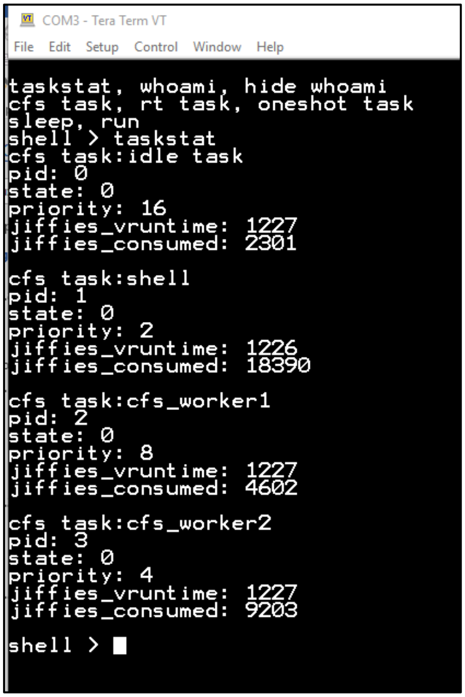
4.7.5 Test of CFS Scheduler

CFS scheduler assigns processor time to a task based on the task's priority. Fairness is measured by virtual runtime determined by task's priority. SLOS sources are branched with the name of *'SLOS_CH4'* for chapter 4. To run the test, run a *git checkout -b your_branch_name SLOS_CH4* command in the SLOS root directory. Build the sources and package it with *petalinux-package* command as described in chapter 2.3.7. The bare SLOS creates two default complete fair tasks in its bootup sequence: cpuidle task and shell task. After finishing bootup, pressing enter key in the serial terminal shows a shell task's command prompt. The shell task gets user command string 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 the shell task. In the shell command prompt, enter *cfs task* command then shell task will create two more cfs worker tasks. Currently, all these tasks don't do any meanful work, just for test purpose except shell task.

Now, we create 4 tasks: cpuidle task, shell task, 2 worker tasks. Let's measure the CFS scheduler's fairness among these 4 tasks. After adding two more tasks, run *taskstat* command in the shell prompt. This command should display all tasks' runtime statistic information like a figure 4-23. This command traverses the tasks' linked list and gets the statistics information of each task. Statistics are task *pid,* task's current state (TASK_RUNNING, TASK_WAITING), priority, virtual_runtime, and total jiffies_consumed by the task. We can calculate the task's absolute runtime with jiffies_consumed value, and measure the fairness with jiffes_vruntime value of each task.

In figure 4-23, the total processor runtime is 344,960msec ((2301 + 18390 + 4602 + 9203) * 10msec) which is equal to 34,496 sched_timer ticks. Since the processor time of each task is proportional to each task's priority, we can calculate the expected processor time of each task by a simple multiplication. The Each task's expected sched_timer ticks is calculated as

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



```
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 > █
```

Figure 4-23: Task statistics by running '*task_stat*' command

This is the processor time that each task should have for the fairness. The error difference between real consumed sched_tick (jiffies_consumed) and its expected time is quite small: The biggest one is under 1%. Notice that the virtual runtimes of all tasks are quite same. The higher priority of a task is, the slower its vitual runtime gets. This means the fairness of CFS scheduler-the virutal runtime of all tasks are same. The test result is summarized in table 4-3. Each task's runtime is quite predictable in CFS scheduler and we can measure the performance of CFS scheduler by comparing them with a real number.

	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

Table 4-3: CFS Scheduler Test

4.8 Realtime Scheduler

As mentioned in the first chapter, SLOS is not a commercial operating system but a hobby operating system. Then, why don't we touch some realtime OS features out of curiosity? SLOS has a *soft realtime* scheduling feature. For this, SLOS implements a preemptive context switching in scheduling and does its best efforts to meet the deadline of a realtime task. The preemptive context switching was described in chapter 4.6.4. Unlike CFS tasks, realtime task doesn't share the its time quantum with other tasks. Instead, each realtime task has its own timer tick (has its own timer in the timer framework) and after completing its job, a realtime task must *yield* the processor to the previous task which it preempted before. The time quantum of realtime task should be long enough to finish the job before the end of deadline.

4.8.1 Realtime Scheduling Features

Naturally, SLOS has following two realtime OS features.

1) Preemptive context switching
2) Earliest deadline first in timer framework

SLOS scheduling is preemptive. A timer interrupt can preempt any current task with next earliest deadline task. Check the figure 4-19. When timer interrupt occurs, and if a scheduler determines which task is the most urgent task, then it switches the content of CONTEXT_MEM. When timer interrupt returns, the processor's pc register points to the next task's context which has the urgency. We can call this preemptive because next task preempts current task's running.

The red-black timer tree has a property of *EDF (Earliest Deadline First)* feature. Every realtime task has its own timer in the timer tree. The sched_timer itself is one of the realtime timer which has a 10msec time deadline. If we add other realtime task, a new realtime timer is added into the timer framework. One of the timer framework's job is maintaining the most urgent timer in its left-most node. This timer has information on timer type (realtime timer or sched timer), handler pointer (CFS scheduler or realtime task body) and so on. In figure 4-16, we already verified that SLOS's timer framework works as an EDF scheduling. What happens if this timer framework misses the deadline is non-deterministic for now. The timer framework just set that task is the most urget task and that task will run next time.

In my opinion, the strict real, hard realtime scheduling is feasible in hardware such as FPGA. Realtime doesn't mean the fastest program, it means a deterministic program. But modern computer architecture with multiple levels of caches makes it impossible to predict strictly the softwares running. In addition, the environment of high-level operating system that allows a buggy program makes realtime scheduling harder. If a system needs a hard realtime, a dedicated hardware such as FPGA could be a better choice.

4.8.2 *yield()*

Every CFS task is running through its whole time quantum once it is scheduled. In other words, if a CFS task is scheduled to run, it occupies the processor through the duration of its timer duration if it is not preemptied by the realtime task. Since the sched_tick is 10msec in SLOS, CFS task runs at least 10msec before CFS scheduler picks another task which is the most suffering task from unfairness.

On the contrary, the realtime task has its own timer tick in the timer framework. The realtime task can use this 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 before the deadline, i.e., it missed the deadline, then it increases its missed count by one. So, the timer tick duration must be set correctly when adding a new realtime task. The duration must have enough margin on the work load of the realtime task.

If the realtime task successfully finishes its job before deadline, it must yield the processor to the task which was preempted by the current realtime task. To return to the preempted task, struct task_struct structure has a yield_task pointer to the preempted task. The timer interrupt service routine updates this pointer variable when switching context between current task and next task. If next task is a realtime task, current task is set as a task pointed by the yield_task pointer of the realtime task (next task). Returning to the previous preempted task is done by *yield*() function. yield() function switches context between current running task and previous preemptied task. The yield() function uses a *switch_context_yield (current, yield_task)* as below.

```
switch_context_yield:
    mov     r12,      r0
    stmia   r12!,     {r0-r11}
    str     sp,       r[12],    #4
    str     lr,       r[12]     #4
    str     lr,       [r12]
    mov     r12,      r1,
    ldmia   r12!,     {r0-r11}
    ldr     sp,       [r12],    #4
    ldr     lr,       [r12],    #4
    mrs     r1,       CPSR
    bic     r1,       r1,       #IF_BIT
    msr     CPSR_c,   r1,
    ldr     pc,       [r12]
    nop
```

This routine gets argument [r0] for the address of current task's TCB and [r1] for the address of yield_task's TCB. 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 ~ line 10) is restoring the cpu registers from the address [r1] which is the yield_task's TCB address. These routines are just load / store instructions between CPU registers and TCBs. The remaining part is enabling the interrupt by clearing the *I, F* bit in CPSR register. This is because when entering the yield() function, it disables the interrupt to prevent other tasks interrupt while context switching.

From SLOS v2.0, in addtion to the RT tasks, the yield() function is also used to the *msleep(), usleep()* functions for the CFS task. When currently running CFS task doesn't need to run any more, it can go to the *waitq* by calling one of the sleep functions. The task calling the sleep function, its task state is changed to *TASK_WAITING* state, it goes to waitqueue and yield current CPU to the yield_task. This blocking task will be back to *TASK_RUNNING* when the sleep timer is expired.

4.8.3 Realtime Task Latency

Realtime scheduling means a deterministic running a task, which means it should guarantee the latency of running a next task. We can measure the SLOS's realtime scheduling latency as in figure 4-24.

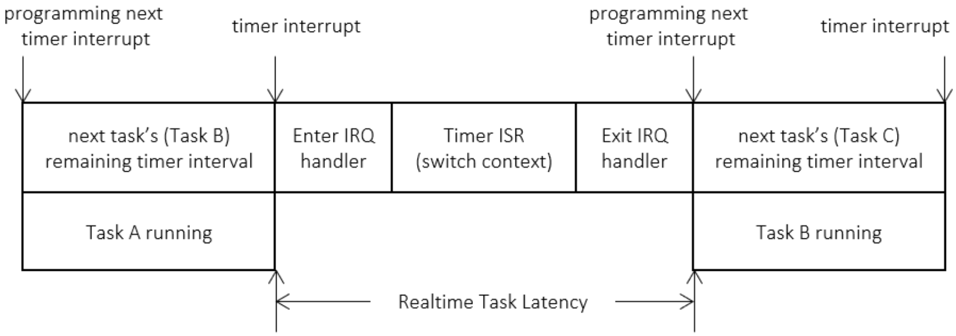

programming next timer interrupt
timer interrupt
programming next timer interrupt
timer interrupt
next task's (Task B) remaining timer interval
Enter IRQ handler
Timer ISR (switch context)
Exit IRQ handler
next task's (Task C) remaining timer interval
Task A running
Task B running
Realtime Task Latency

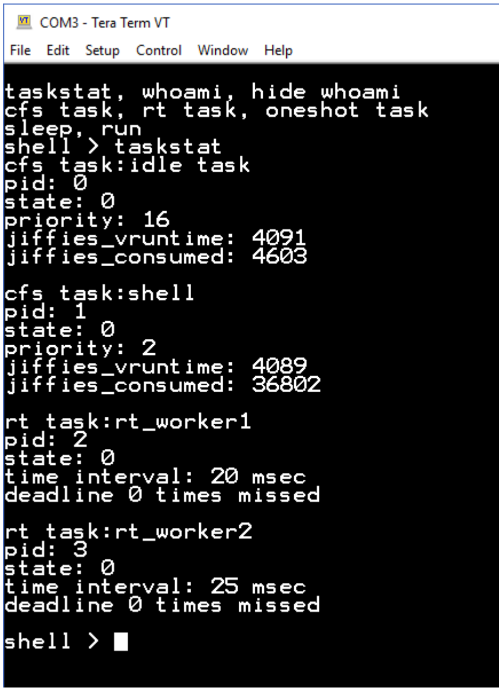
Figure 4-24: Measurement of realtime task's latency

Unfortunately, as of time when this book is written, timer framework in SLOS doesn't have an ability to measure this latency. The timer framework measures only the elapsed time which is equal to the time difference between programming the timer interrupt and real timer interrupt: Task A or Task B running time. To measure the latency correctly, SLOS must have another timer. Zynq processor has more timers such as 64bit Global Timer and 16bit

Triple Timer Counters (TTC). One of these timers could be used to measure the latency. This is not yet done, and can be added later.

4.8.4 Test Realtime Scheduler

To test scheduling of realtime tasks, run a *rt task* command in a shell prompt, then the shell task creates 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. Since SLOS is soft realtime, even though they missed their deadline, SLOS just increases the missed count of the them and keeps going. Realtime task 1 and 2 do a simple, dummy thing that increases a variable in a for loop for a moment, then yield the processor to the task pointed by struct task_struct *yield_task. Check whether the task 1 and 2 miss any deadline by running *taskstat* command as figure 4-25.



```
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: 4091
 jiffies_consumed: 4603

cfs task:shell
pid: 1
state: 0
priority: 2
 jiffies_vruntime: 4089
 jiffies_consumed: 36802

rt task:rt_worker1
pid: 2
state: 0
time interval: 20 msec
deadline 0 times missed

rt task:rt_worker2
pid: 3
state: 0
time interval: 25 msec
deadline 0 times missed

shell > █
```

Figure 4-25: Scheduling test for realtime tasks

Since the jobs in task 1, 2 are pretty small compared to the deadline, you shouldn't see any missed deadline as in figure 4-25. The realtime task doesn't have a priority, sched_entity and jiffies for sched_timer tick. Instead, it just has a missed count for its runtime information.

If multiple realtime work has a same deadline, we can use a priority information for picking up a more urgent task. Since the cpuidle task and shell task are always running by default, they are also shown in figure 4-25. Their virtual runtime is still pretty same but the jiffies_consumed value that is the real runtime is proportional to their priority. The realtime task, rt_worker1 and rt_worker2, doesn't miss their deadline in this test.

4.9 Putting it altogether

4.9.1 Kernel Main Function Sequence

The main() function, SLOS's kernel entry point, initializes the modules for process management: GIC and timer framework. Then, it creates the shell task and cpuidle task. Shell task and cpuidle task are SLOS default CFS tasks. Kernel main routine follows the sequence in figure 4-26.

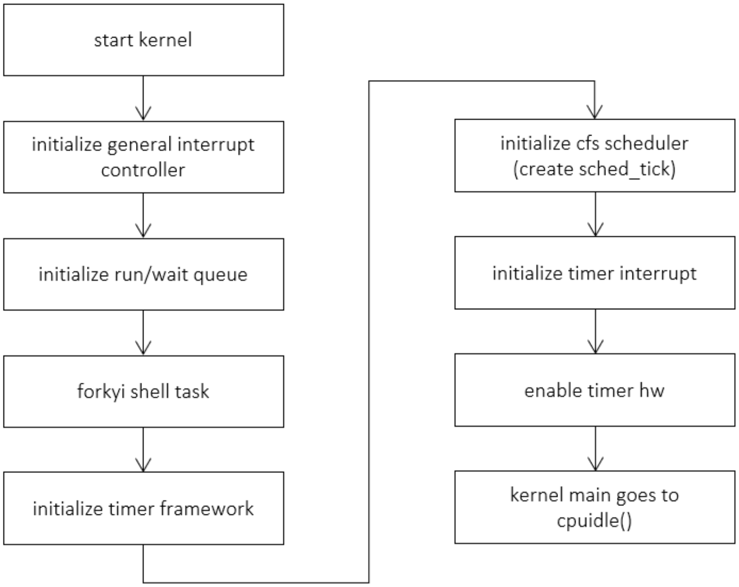

start kernel
initialize general interrupt controller
initialize run/wait queue
forkyi shell task
initialize timer framework
initialize cfs scheduler (create sched_tick)
initialize timer interrupt
enable timer hw
kernel main goes to cpuidle()

Figure 4-26: SLOS initialization sequence in main() function

The initialization routine in main() function will be updated later as more features (*memory management, storage management, custom hardware design*) are added. The sequence in figure 4-26 is only for the initialization of the process management.

While booting, the kernel main initializes the GIC hardware and private timer hardware. Those two hardwares are all we need to implement for the process management. The main() function also creates the shell task by using forkyi(). Even after the shell task is forked, it doesn't yet run. The shell task is just put into the runqueue, but the scheduler doesn't schedule anything until timer beat starts. main() function initializes timer framework, CFS scheduler, enables the timer hardware to start the scheduler and falls back to its infinite loop of cpuidle task.

4.9.2 Test of All Tasks

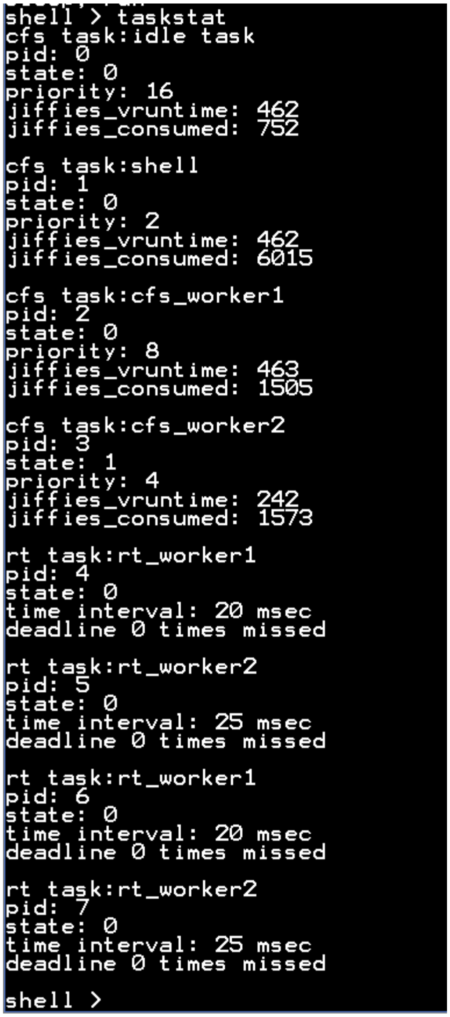
After completing the bootup, kernel main() routine turns into to the cpuidle task. SLOS has only shell task and cpuidle task after bootup finishes. cpuidle task has PID (Process ID) 0 because it is the first task running after bootup, and shell task has PID 1, the next task of cpuidle.

After checking out the SLOS sources with branch *'SLOS_CH4',* build the sources and make the BOOT.BIN as described in chapter 2.3.7. Insert the SD card to the Zynq evalution board and boot up the board. Connect a serial terminal and properly set the terminal's configuration. After hitting enter key, you should see the shell task's command prompt.

We can add two more CFS tasks, two realtime task and oneshot task by using a shell task's command in the prompt. SLOS has a limit in SVC mode stack size which is 64KB. Task's stack map is predefined in the system memory map described in figure 4.2. Since each task has 4KB stack size, SLOS can have up to 16 tasks at most. Input *cfs task* and *rt task* command after SLOS completes its bootup. SLOS process management handles the CFS tasks and realtime tasks through its timer framework. Then, after a while, run *taskstat* command to list up the status of current tasks. An example of task statistics is illustrated in figure 4-27.

In the figure 4-27, two additional CFS tasks and four realtime tasks are added. Notice that the value of cfs_worker2's state is 1 which means it is in TASK_WAITING state waiting in a waitqueue. Adding more CFS tasks makes the shell task's response slow because all CFS tasks share one sched_timer tick. The more CFS tasks there are, the less possibility of sched_timer tick that the shell task can get. This results in a less responsiveness of shell task.

The tasks' statistics information in figure 4-27 looks good: all CFS tasks still has fairness (same virtual runtime but priority-proportional jiffies_consumed) and all realtime tasks doesn't miss their deadline. When cfs_worker2 task is dequeued to waitqueue, the CFS scheduler correctly update the priority sum and each running CFS task's virtual runtime as depicted in figure 4-18. If the waiting task (cfs_worker2) is enqueued to the runqueue again, CFS task will correctly update the priority sum and virtual runtime of all tasks.



```
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 >
```

Figure 4-27: Putting it altogather

4.10 Power Management in Process Management

Power management in mobile device is so important these days that the battery life and *Day of Use (DoU)* are key performance factors of those products. Power management is related to the whole system including chipset fabrication, peripheral hardware, device drivers, applications and so on. In a smartphone, the most power consuming module in the system is a processor. Then display module follows the next. As the display size gets bigger, the portion of power consumed from display also gets higher, but still the dominant power including thermal is dissipated from the chipset processor. If you have any interests regarding a chipset power, you should keep below equation in mind.

$$Power_{CMOS} = Power_{dynamic} + Power_{static} + Power_{short}$$
$$= CV_{DD}^2 f_{sw} + I_{leakage}V_{DD} + I_{short}V_{DD}$$

Equation 4-2: Power consumption in a CMOS

This is an atomic power consumed in one *CMOS* circuit in figure 4-28. A CMOS in figure 4-28 is an inverter which represents one digital bit inverted from the input. Theoretically, this CMOS design consumes the power only when the signal is changed. But practically, there are additional power consumption coming from leakage current, and instantaneous short circuit current. A chipset processor is composed of milions of this CMOS. So, knowing the power consumption in this CMOS is basic to optimize the power in the chipset.

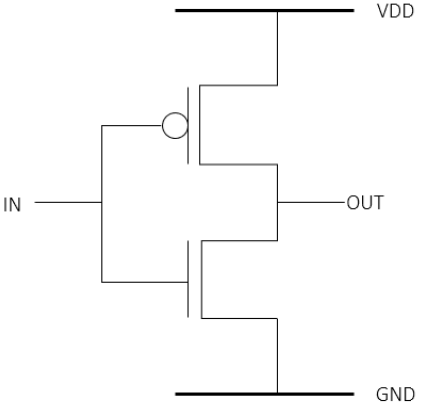


Figure 4-28: An inverter CMOS to store a digital 1 bit

As you can see in equation 4-2, the chipset power is composed of dynamic power*,* static power and short circuit power. Noramlly, the major power consumption comes from dynamic power (70 ~ 80%), and static power (~20%), and short circuit power (~5%). As the chipset fabrication gets narrower, the static power gets bigger but still the dynamic power is dominant.

1) *Dynamic Power*

   This power consumption happens only when CMOS switching occurs. CMOS switching means the input/output of CMOS changes. This is why we call it 'dynamic'**.** This power is mostly dependent on the VDD voltage and switching frequency. *C* in equation 4-2 is a constant dependent on the fabrication (*W/L)*, mostly capacitance.

2) *Static Power*

   This power consumption comes from a leakage current from VDD to GND.

Theoretically, CMOS doesn't consume power if the input doesn't change. But practically, there is a leakage current and this results in a static power dissipation. As the fabrication process gets narrower, the switching speed gets faster and CMOS works in a lower VDD. But this makes the leakage current become bigger.

3) *Short Circuit Power*
When CMOS changes its state, there is a short period when both PMOS and NMOS are closed simultaneoulsy. This makes an instantaneous short circuit and a short curcuit current flows. This is not that much, but it still dissipates a little power.

In the power management for software engineer, dynamic power is our major concern. Software engineers working on kernel and device driver also develop many good schemes to reduce the power consumption in the chipsets. Hardware engineer can reduce the dynamic power by controlling the VDD and switching frequency. Power management is really expanded into the areas between hardware and software.

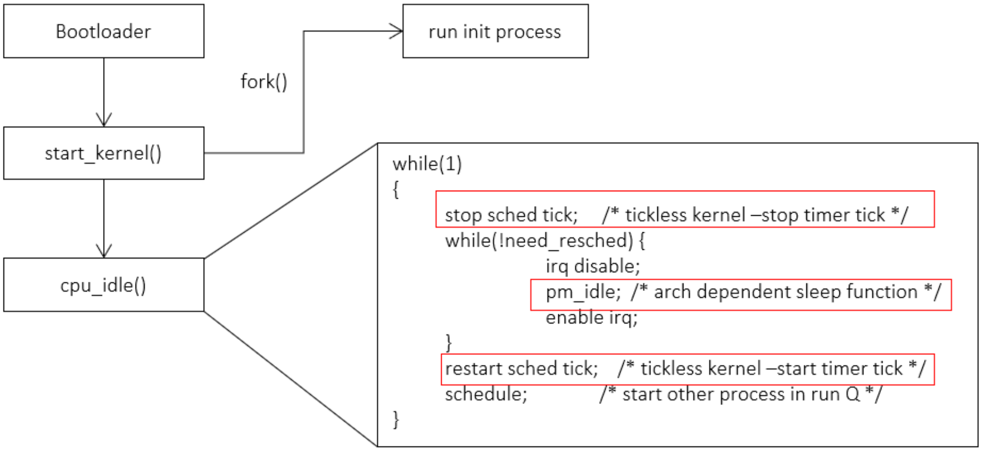
Modern chipsets allow the operating system to control the VDD and switching frequency in real time based on the processor's work load. Since process management in operating system knows the work load or even it predicts future work load also, process management is closely connected with the dynamic power optimization. In this chapter, this book will touch a basic, software-wise power optimization related to the process management.

4.10.1 CPUIDLE - Sleep it as much as possible

As we already described earlier in chapter 4.7.3, cpudile task plays an important role in power optimization. cpuidle task in SLOS does almost nothing. But high-level OS makes the processor sleep, clock gating and turns off some power rails in cpuidle task. Running cpuidle task means there is nothing to run in the processor. So, that is a proper time to make the processor sleep. When processor goes to sleep, the operating system can manipulate the voltages (powers) to the chipset subsystems. If operating system determines that the core processor can be off, it turns off the power rail to the core. If other subsystem (e.g., bus) in the chipset goes to sleep, power rail to that subsystem also can be off. Some chipsets have levels in turning off power rails. Operating system chooses the sleep level based on the scheduler information, and turns off the power rails accordingly. For example, if there is no job to run for 1sec, the operating system could make the processor go to deep sleep and collapse whole power of the processor. This saves the power the most. If there is no job to run for 100msec, the operating system could turn off some part of power rails to the processor. This saves the power lesser but still better than nothing. Since the processor wake up time in deep sleep takes more than in light sleep, the scheduler information used to determine the level of processor sleep. This scheme saves huge amout of power. Notice that when you just look at your smartphone without doing anything, even the display is on, the processor could be in deep sleep and the power to the processor could be collapsed.

Another thing I want to mention on cpuidle is that a *tickless kernel.* As the title of this chapter says, the processor must be in sleep as much as possible to save power. Waking up the processor from its sleep mode results in power consumption. Normally, there is two sources to wake up the processor out of sleep: interrupt and timer. Timer is also an interrupt but let's use it differently here because timer interrupt is related to OS scheduler. Do you remember the periodic sched_timer tick for CFS tasks? This timer interrupt occurs every 10msec even though there is no work (task) to run from the processor. If timer interrupt wakes up the processor, then the processor figures out there is no work to run and falls back to sleep again. This happens every 10msec. Moreover, if the sleep time is small, then the processor doesn't go to deep sleep mode which save the most power. This is too bad for power optimization. So, the kernel engineers devise a *tickless kernel*. A tickless kernel turns off the periodic sched_timer tick until next task runs when it goes to sleep. For example, if next task is supposed to run in 500msec, the cpuidle turns off the 10msec *sched_timer* tick when processor goes to sleep. This makes the processor go to deep sleep, and saves more power by not in-and-outing from the sleep.

Linux has all these features. Figure 4-29 shows the cpuidle in Linux. After it boots up, the kernel main entry point, start_kernel() function, turns into cpu_idle(). The cpu_idle() in Linux does two important things. First cpu_idle() turns off sched_tick before it goes to sleep and turns back on after exit sleep. Second, when it goes to sleep, it calls a platform dependent sleep function. The pm_ilde() is a function pointer to the platform dependent sleep implementation. Each chipset vendor implements their sleep code for their custom chipset into here.



```
while(1)
{
        stop sched tick;    /* tickless kernel –stop timer tick */
        while(!need_resched) {
                        irq disable;
                        pm_idle;  /* arch dependent sleep function */
                        enable irq;
        }
        restart sched tick;    /* tickless kernel –start timer tick */
        schedule;            /* start other process in run Q */
}
```

Figure 4-29: Linux cpu_idle() for idle power saving

4.10.2 CPUFREQUENCY - Keep it slow as much as possible

This title sounds weird - do we have to intentionally lower the cpu processor performance to save power? The answer is *"Yes or No".* You should do this in some cases. This is why people say there is a tradeoff between processor performance and processor power saving. The more performance, the more power are needed and the more thermal is dissipated. Sometimes, there is a conflict between power team and performance team because of this.

To save the processor's runtime power, the dynamic power factor in equation 4-2 must be taken care of. There is a simple chain reaction in the dynamic power. As there are more and more works for the processor, the processor's running frequencey goes up. This results in a faster switching frequency in CMOS. A higher switching frequency in CMOS needs a low delay in the CMOS gates. This CMOS gate delay is inverse-proportional to the VDD: higher VDD means faster switching reaction in the gate (There is also an equation for the relation between gate delay and VDD, but I will not write it down because we already have too many equations). So, faster switching frequency needs a higher VDD. This chain is summarized in figure 4-30.

CPU has the ability to change its running frequency and core voltage. The operating system dynamically sets the CPU processor frequency and core VDD in realtime based on the work load. For example, let's assume the CPU supports 500Mhz, 1Ghz, and 2Ghz for its running frequency. Currently, it works in 500Mhz. If there are many works in its runqueue, this CPU can't sleep and runs 100% to finish its work load. Then OS determines current CPU frequency is not enough for current work loads and then increase the CPU frequency to the max frequency, 2Ghz, to finish up its work as soon as possible. When OS sets the CPU frequency, there is an associated CPU core voltage, VDD, and OS also sets the proper VDD. In this state, CPU consumes much power because the frequency and voltage both go up. After finishing those work with its best performance and worst power consumption, the CPU usage goes down, then the OS makes CPU frequency be lowered and sets lower VDD as well. This low performance has lower power consumption. This is how the processor's runtime power saving works.

In addition to this processor power saving, there are many other power saving schemes in modern operating system: clock gating, device driver power management, application level wakeup reduction and so on. Working power management needs a system-wide knowledge from hardware, chipset, operating system to application software.

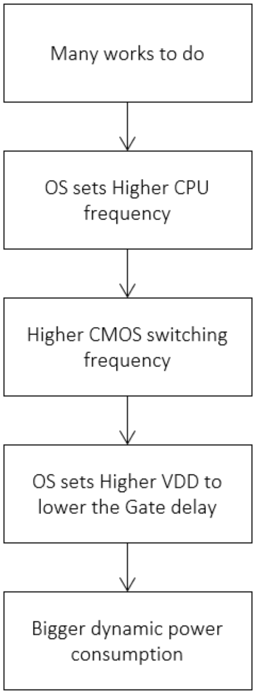

Many works to do
OS sets Higher CPU frequency
Higher CMOS switching frequency
OS sets Higher VDD to lower the Gate delay
Bigger dynamic power consumption

Figure 4-30: A chain reaction for dynamic power consumption

4.11 Summary

In this chapter, we built two schedulers: CFS scheduler and Realtime scheduler. Before delving into the implementation of scheduler, we need to define or develop a prerequiste for them: Generic Interrupt Controller (GIC), Timer framework, Task Control Block (TCB), Timer Interrupt Handler and Context Switching. We also need to know a little bit about the hardware portion of ARM such as CPU private timer, ARM registers, ARM processor modes, ARM ISA (Instruction Set Architecture) and so on. Some of these are covered in chapter 2.5, ARMv7 architecture introduction. Spinlock implementation introduces new ISA (LDREX, STREX) in this chapter. The ARM exclusive monitor was also covered for the task synchronization.

SLOS process manager supports CFS scheduler and Realtime scheduler simultaneously. Actually, the CFS scheduler also uses one of Realtime timer but the time quantum is fairly shared among all CFS tasks. We verified the fairness of CFS scheduler by experiment in chapter 4.7.4.

Tasks in SLOS has its own running states and process management has runqueue and waitqueue for this. Tasks has some other information such as PID, statistic information.

SLOS also has a preemptive context switching to support its realtime scheduling. Current running task can be preempted with the task which is scheduled by the Earliest Deadline First (EDF) scheme.

The coexistence of CFS scheduler and Realtime scheduler is verified in chapter 4.5.4, the timer framework chapter. Timer framework is core part of SLOS's process management. Timer framework is using red-black tree to manage the timer nodes created for both schedulers. It returns the timer having the earliest deadline.

In chapter 4.9, we checked the process management with the real hardware (Zynq7000 evaluation board).

References

[1] Xilinx User Guide: UG585 Zynq-7000 All Programmable SoC Technical Reference Manual

[2] https://en.wikipedia.org/wiki/Coprocessor

[3] ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition

[4] ARM Generic Interrupt Controller Architecture Specification version 1.0

[5] https://github.com/torvalds/linux/blob/master/Documentation/rbtree.txt

[6] Remzi Arpaci-Dusseau, Andrea Arpaci-Dusseau, *Operating Systems: Three Easy Pieces,* Version 0.92, Arpaci-Dusseau Books (2018)

[7] ldrex, strex:
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0008a/ch01s02s01.html

[8] Exclusive Monitor:
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0008a/CJAGCFAF.html

[9] ARM Cortex-A Series Programmer's Guide

[10] ARMv7-M Architecture Reference Manual

5 Memory Management

5.1 Introduction

In chapter 4, we've already done a simple memory management such as build memory map and a heap memory management implementation. But these are temporary memory management and valid only for process management. In this chapter, we are going to develop simple but essential features in modern OS's memory management. It doesn't cover all features in memory management but it can give you a basic intuition for core memory management concepts.

The process management mostly focuses on the core processor and its virtualization. With memory management, the SLOS can be extended to the system-wide level such as memory proctection between applications, page swap and so on.

After memory management is added, the SLOS top view looks like figure 5-1. Memory management blocks are newly added into previous figure 4-1, and they are grayed blocks in figure 5-1. Memory management in SLOS has virtual memory, page table and demands new pages when page fault happens (demand paging). Followings are basic and essential parts of SLOS memory management.

1) Data Abort Handler in Exception Vector
   A new exception handler is added for memory management. A data abort handler runs when page fault happens.

2) Virtual Memory Manager
   Virtual memory manager builds a page tables and enables *MMU (Memory Management Unit).* Previous kmalloc()/kfree() functions are supposed to be changed to use the virtual memory manager.

3) MMU Page Table Walk
   Page table walk is used to translate the virtual address and find the physical address of a memory access.

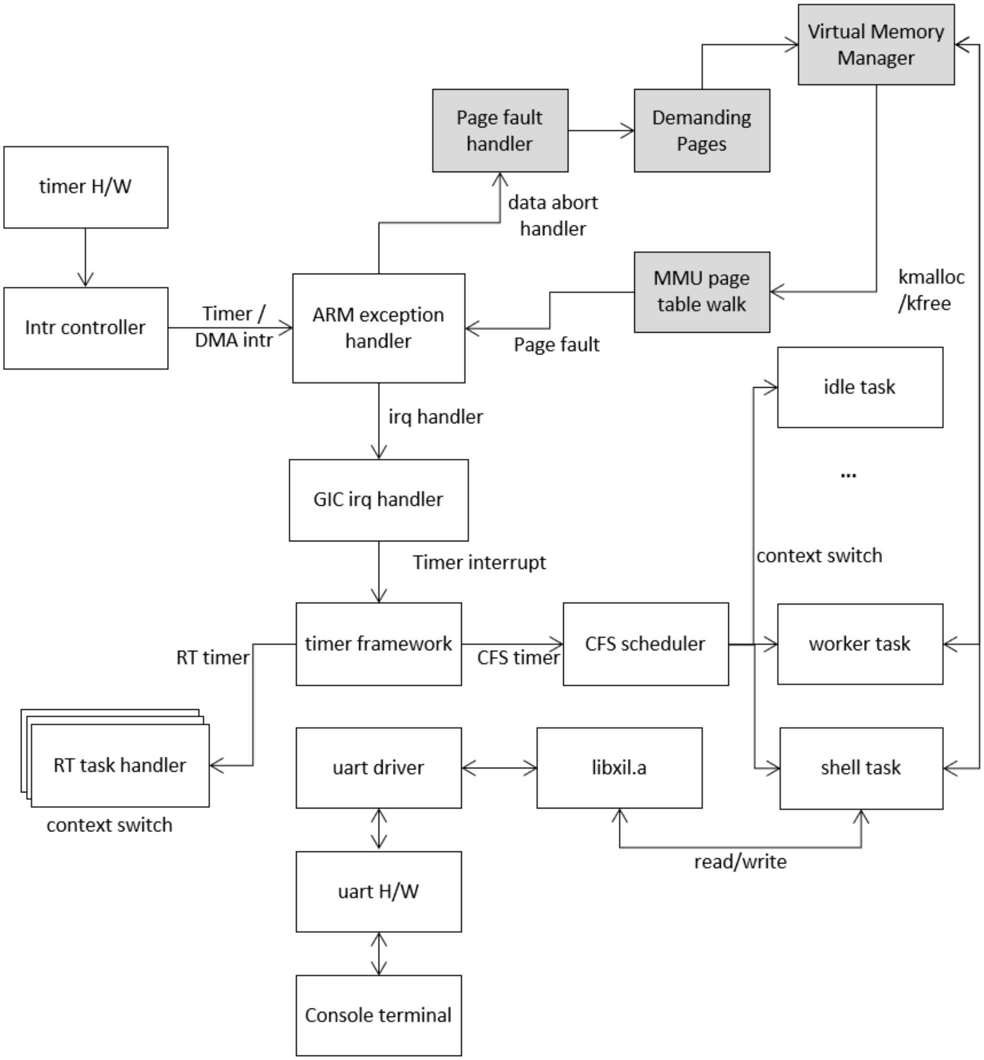


Figure 5-1: SLOS top view when memory management is added

SLOS has new files or some changes from chapter 4 for memory management. Here are summaries of changes.

1) *kernel/exception/init_mm.S*
   This is a new file added for the initialization of memory manager. The memory initialization code is located in *SSBL (Second Stage BootLoader)* section together with page table initialization routine. The SSBL initializes the MMU, page translation tables, enables MMU and jumps to the reset vector. This file also has the entry point of second stage bootloader.

2) *kernel/exception/kernel.S*
The exception vector handler has another implementation for *data abort handler* which has offset of 0x10 from the vector base address. The data abort handler is used to implement a *page fault handler.* After this chapter, the kernel.S file has more of exception handler implementations for *reset vector handler, data abort handler* and *irq handler.*

3) *kernel/exception/faults.c*
This new file contains functions which are called from the data abort exception handler. This file checks the abort type and if it is a page fault, it calls a proper page fault handler routine. If it is a different type of a fault, it makes the SLOS kernel spin forever and never returns from the abort routine.

4) *kernel/core/frame_pool.c*
This new file has the implementations about the physical memory management. SLOS memory manager views the physical memory with 4KB *memory frame* blocks. This frame size is same as the size of *small page* which is used by the MMU.

5) *kernel/core/page_table.c*
This is a new file for maintaining the *page table*. It can create a new entry to page table when page fault happens *(demanding pages)* in the fault handler. This file has routines to free unused pages and to delete the entry for that page. It also initializes the kernel's *page table base address*.

6) *kernel/core/vm_pool.c*
This file has the implementations of *virtual memory pool management*. Virtual memory pool manager has the *virtual memory region descriptors*, the implementations of allocating, releasing the virtual memory regions. The virtual memory pool manager actually doesn't allocate the physical memory at the moment of request. Rather it just allocates virtual memory region descriptor in its meta data region and allocate later when there is a real access to that memory region. We can call this a *lazy allocator*.

7) *kernel/core/mm.c*
The *mm.c* has init_pgt() function, a translation table initialization function, and heap memory kalloc() and kfree() functions. kmalloc() function was already used in chapter 4 with its simplest implementation. In this chpater, kmalloc() function is properly implemented by using a virtual memory manager. The init_pgt() function

along with init_mm.S is mapped to the .ssbl section which is a second stage bootloader.

8) *kernel/linker/kernel.lds*

The functions in the secondary bootloader are determined by the linker script. Remember that linker script's role is mapping the input sections to output sections. From the memory management, SLOS uses a virtual address. SLOS is still loaded into 0x0010_0000 physically but its virtual address starts from 0xC010_0000. There is a 0xC000_0000 offset between load address and virtual address. So, the *VMA (Virtual Memory Address)* is different with *LMA (Load Memory Address)* after enabling the memory management. We shortly mentioned on this in chapter 3.3.1 in short description of linker script. This discrepency is solved in the linker script file.

9) *kernel/inc/mem_layout.h, frame_pool.h, page_table.h, vm_pool.h, mm.h*

These are header files having declarations of the functions, structure definitions and macro definitions for memory management. Notice that init_pgt() function is declared as below in mm.h. This routine makes the init_pgt() function placed into the *'PGT_INIT'* section. The init_pgt() function is declared as below using the *__attribute__* keyword.

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

SLOS sources for memory management has branch name *'SLOS_CH5'* and can be checked out by following command.

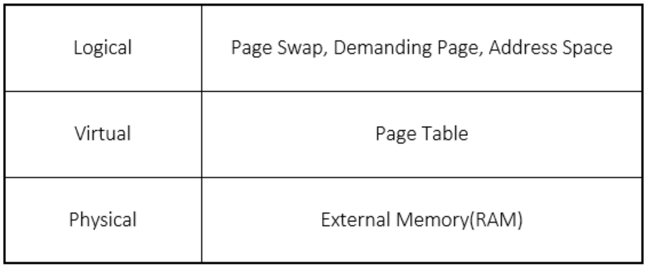
git checkout -b your_local_branch_name SLOS_CH5

5.2 Address Space

5.2.1 *Logical Address, Virtual Address, Physical Address*

Operating system sees one memory location with different views. We can call each different view on memory as an address space. The address space is analysed in a similar way that is used by the previous *layered model* in this book which was mentioned several times before. We used a physical layer as referring to the hardware itself. So, the physical address is used to refer to one location in the physical layer. The virtual layer is used to virtualize the physical hardware resource - it is RAM memory in memory management. We name this virtual layer as virtual address that is managed by MMU. So, the MMU takes care of virtual address. Logical address is the address used in logical layer which is the address used by core processor. Figure 5-2 shows the relation on different address spaces.

Figure 5-2 shows which layer uses which address space. The virtual address layer is optional. If MMU is not enabled, there is no virtual address layer. In this case the logical address is exactly same as physical address; fancy logical features can't be used. If MMU is enabled, the virtual address space abstracts the physical address space. In this case, the logical address is same as virtual address but the virtual address needs a translation to go to physical address space. This translation is processed by the MMU. Once there is a virtual layer to abstract the physical layer of a memory, many good features of logical layer can be enabled.



Logical	Page Swap, Demanding Page, Address Space
Virtual	Page Table
Physical	External Memory(RAM)

Figure 5-2: Layered model of address

MMU uses a translation table that maps the virtual address to the physical address. This translation table is also saved in some area of the memory and one of the MMU register contains the start address of this table. There is a question in my mind when I first work on the translation table walking. Is this start address of a translation table a physical address or a virtual address in this case? The start address of translation table is the first place to translate the logical address. If this is a virtual address, how can the MMU translate the start address? It made a confusion to me at that time. The answer is simple. Virtual address is used in CPU, same as logical address. The start address of translation table is used by MMU. So, the answer is clear - it's a physical address. MMU can use this address without translating it.

5.2.2 *Memory Mapped IO (MMIO)*

ARM processor uses a *Memory Mapped IO (MMIO)* for its accessing of an IO operations. A memory mapped IO is one of the methods performing the input/output between CPU and its peripheral devices. There is another *Port Mapped IO (PMIO),* but we will not touch the concept of this because we don't use it.

Memory Mapped IO uses one unified address space to address both memory and IO peripheral devices. The memory and registers of IO devices are mapped (associated with) by the processor's memory address values. The CPU doesn't know whether current address is for an access to a memory region or to the registers in an IO peripheral device. One merit of memory-mapped I/O is that, by discarding the extra complexity that port I/O brings, a CPU requires less internal logic and is thus cheaper, faster, easier to build, consumes less power and can be physically smaller; this follows the basic tenets of reduced instruction set computing, and is also advantageous in embedded systems. The other advantage is that, because regular memory instructions are used to address devices, all of the CPU's addressing modes are available for the I/O as well as the memory, and instructions that perform an ALU operation directly on a memory operand (loading an operand from a memory location, storing the result to a memory location, or both) can be used with I/O device registers as well [1].

Since the CPU recognizes the address for IO device as a memory address, we also need to consider the cache operation in this IO memory access. If cache is located in-between the CPU and IO devices, the access order is not guaranted. This means IO operation is not guaranteed; the access to IO device must be ordered. Who want to see the “dlrow olleh” in the terminal screen when he types “hello world”? To solve this issue in IO memory access, when setting up the translation table, we can set some properties on the memory region. Some address for IO device must have special properties such as *Non-Cacheable,* or *Strongly-Ordered Memory.* We will cover this later in this chpater.

When we say 32bit processor architecture can access 4GB memory, it doesn’t mean that processor can have 4GB of RAM memory in memory mapped IO architecture. The 4GB address space should be shared between RAM memory and other IO devices. For example, Zynq7000 chipset assigns 2GB address (0x4000_0000 ~0xBFFF_FFFF) to *AXI Master* interface IO operation. This region can’t be used by external memory address. But most of the embedded system doesn’t need the whole 4GB RAM memory. In addition, ARMv7-A architecture has a *Large Physical Address Extension (LPAE)* that provides an address translation up to 40bits which is almost infinite amount of memory for an embedded system.

5.3 ARM Memory Management Unit (MMU)

A *Memory Management Unit (MMU)* is a special hardware block performing the virtualization of physical memory. After MMU is enabled, all addresses in the program running in the CPU are recognized as a virtual address. In order to access a specific address location in physical memory, the virtual address must be translated by the MMU. MMU doesn’t generate a virtual address, but it translates the virtual address to the physical address. It is placed between core processor and external memory as in figure 5-3.

Generally, the MMU does two primary functions: it translates virtual address into physical addresses, and it controls memory access permissions. The MMU hardware required to perform these functions consists of a *Translation Lookaside Buffer (TLB), access control logic,* and *Translatioin Table walking logic.*

The ARM MMU supports memory accesses based on *Sections and Pages.* Sections are comprised of 1MB, 16MB (*Supersection)* blocks of memory. Pages are 4KB *(Small Pages)* blocks of memroy or 64KB *(Large Pages)* blocks of memory. One block of these size is associated with one entry of the translation table.

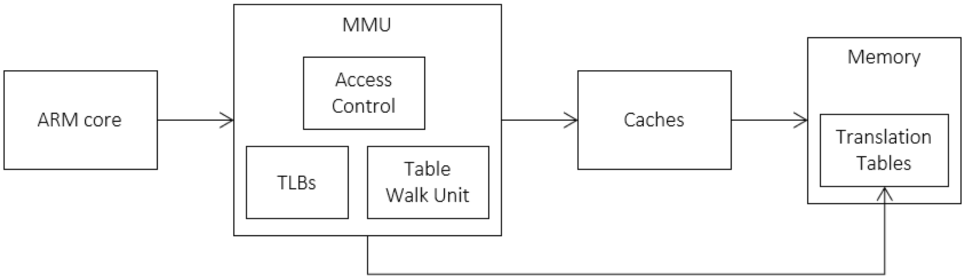


Figure 5-3: ARM core and MMU blocks to access memory

The MMU also supports the concept of *domains:* areas of memory that can be defined to possess individual access rights. The *Domain Access Control Register* is used to specify the access rights for up to 16 separate domains.

The translation lookaside buffer (TLB) caches the virtual address translation for a better performance. Since the translation table itself is placed in the external memory, accessing to this table whenever access to a memory results in a critical performance degradation. So, TLB caches the translation table in the memory for better performance. If the TLB hits a translated entry for the virtual address, the access control logic determines whether the access is permitted. If access is permitted, MMU returns the physical address corresponding to the virtual address. If access is not permitted, MMU signal CPU to generate abort.

If TLB cache miss happens, the translation table walk hardware is invoked to retrieve the translation information from a translation table in physical memory. Once retrieved, the translation information is put into the TLB, possibly overwriting an existing value.

There is a *System Control Coprocessor Register (CP15)* that defines the location of the translation tables, enabling and configuring the MMU. For a detailed description on the coprocessor register CP15, refer ARM reference document [2] part B, *System Level Architecture.*

5.3.1 SCTLR, System Control Register

The System Control Register (SCTLR) provides a basic, important control of the system, including its memory system. The SCTLR register bit assignments look like figure 5-4. Let's have a look at some system configuration bits.

1) AFE, bit[29]
   Access flag enable bit. Possible values of this bit are:
   0: In the translation descriptor, the AP[0] is used for access permissions bit. The full range of access permission control is supported. No Access flag is implemented.
   1: In the table translation descriptor, the AP[0] is used for access flag. Only simplified access permissions are supported.

This bit is used to set how the access permission bits (AP[2:1]) are used.

2) I, bit[12]
Instruction cache enable. This is a global enable bit for instruction cache. Possible values of this are:
0: Instruction caches disabled
1: Instruction caches enabled

3) C, bit[2]
Cache enable. This is a global enable bit for data and unified caches. The possible values of this are:
0: Data and unified caches disabled
1: Data and unified caches enabled

4) A, bit[1]
Alignment check enable. This is the enable bit for alignment fault checking. The possible values of this bit are:
0: Alignment fault checking disabled
1: Alignment fault checking enabled

5) M, bit[0]
MMU enable. This is a global enable bit for the PL1 & 0 stage 1 MMU. The possible values of this bit are:
0: PL1 & 0 stage 1 MMU disabled
1: PL1 &0 stage 1 MMU enabled

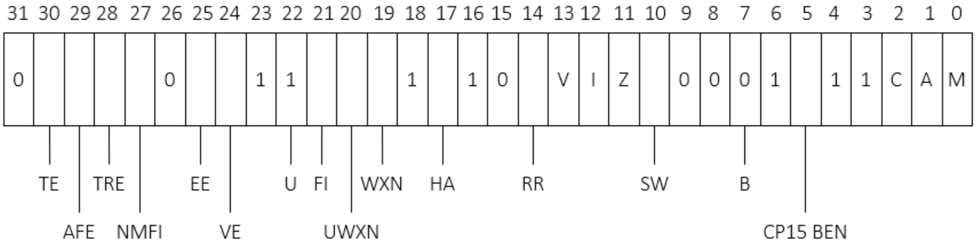


Figure 5-4: System Control Register bit assignment

Refer ARM technical reference manual document for the descriptions on other bits. We set those bits in SCTLR to manipulate MMU and caches. Read/Wrtie to the SCTLR is performaned through CP register as below.

MRC p15, 0, <Rt>, c1, c0, 0 ; Read SCTLR into Rt

```
MCR p15, 0, <Rt>, c1, c0, 0                    ; Write Rt to SCTLR
```

5.3.2 Memory Ordering

We mentioned a little bit before in chapter 4.7.3, cpuidle task and in chapter 5.2.2, memory mapped IO. ARM processor needs a specific configuration of each memory region based on its memory ordering. Following description mostly comes from reference [5].

Older implementations of the ARM architecture execute all instructions in program order and each instruction is completely executed before the next instruction is started. Newer processors employ a number of optimizations that relate to the order in which instructions are executed and the way memory accesses are performed. The speed of execution of instructions by the core is significantly higher than the speed of external memory. Caches and write buffers are used to partially hide the latency associated with this difference in speed. One potential effect of this is to re-order memory accesses. The order in which load and store instructions are executed by the core will not necessarily be the same as the order in which the accesses are seen by external devices.

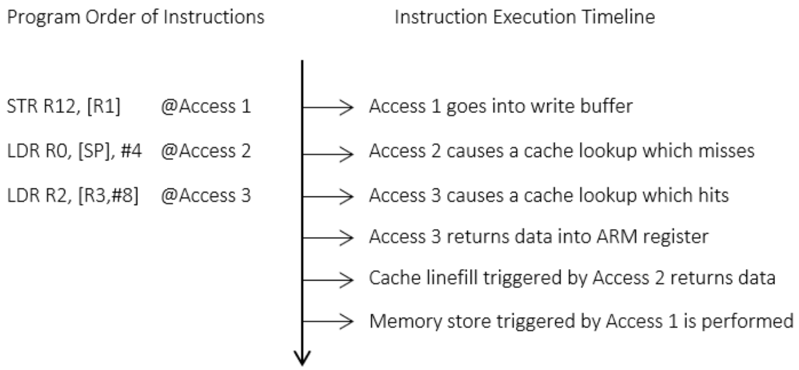

Program Order of Instructions
Instruction Execution Timeline
STR R12, [R1]
@Access 1
LDR R0, [SP], #4
@Access 2
LDR R2, [R3,#8]
@Access 3
Access 1 goes into write buffer
Access 2 causes a cache lookup which misses
Access 3 causes a cache lookup which hits
Access 3 returns data into ARM register
Cache linefill triggered by Access 2 returns data
Memory store triggered by Access 1 is performed

Figure 5-5: Memory ordering example

In Figure 5-5, three instructions are listed in program order. The first instruction performs a write to external memory that in this example, goes to the write buffer (Access 1). It is followed in program order by two reads, one that misses in the cache (Access 2) and one that hits in the cache (Access 3). Both of the read accesses could complete before the write buffer completes the write associated with Access 1. Hit-under-miss behaviors in the cache mean that a load that hits in the cache (like Access 3) can complete before a load earlier in the program that missed in the cache (like Access 2).

It is still possible to preserve the illusion that the hardware executes instructions in the order you wrote them. There are generally only a few cases where you have to worry about such effects. For example, if you are modifying CP15 registers, copying or otherwise changing code in memory, it might be necessary to explicitly make the core wait for such operations to complete. For very high-performance cores that support speculative data accesses, multi-issuing of instructions, cache coherency protocols and out-of-order execution in order to make additional performance gains, there are even greater possibilities for re-ordering. In general, the effects of this re-ordering are invisible to you, in a single core system. The hardware takes care of many possible hazards. It will ensure that data dependencies are respected and ensure the correct value is returned by a read, allowing for potential modifications caused by earlier writes.

However, in cases where you have multiple cores that communicate through shared memory (or share data in other ways), memory ordering considerations become more important. In general, you are most likely to care about exact memory ordering at points where multiple execution threads must be synchronized. Processors that conform to the ARMv7-A architecture employ a *weakly-ordered* model of memory, this means that the order of memory accesses is not required to be the same as the program order for load and store operations. The model can reorder memory read operations (such as LDR, LDM and LDD instructions) with respect to each other, to store operations, and certain other instructions. Reads and writes to *Normal memory* can be re-ordered by hardware, with such re-ordering being subject only to data dependencies and explicit memory barrier instructions. In cases where stronger ordering rules are required, this is communicated to the core through the memory type attribute of the translation table entry that describes that memory. Enforcing ordering rules on the core limits the possible hardware optimizations and therefore reduces performance and increases power consumption.

5.3.3 ARM Memory Ordering Model

ARM is weakly-ordered memory but it also supports 3 types of memory for the control of memory access order. When specific memory region needs to be configured as an ordered access, that region can be marked as strongly-ordered. In this case, memory access is guaranteed to occur in the order they issued. ARM defines three mutually exclusive memory types. All regions of memory are configured as one of these three types:

1) Strongly-ordered memory
   Access is atomic, not cached and speculative access is never performed. A write can complete only when it reaches the peripheral or memory component accessed by the write.
2) Device memory
   Access is atomic, not cached and speculative access is never performed. The order of access at Device memory is guaranteed to correspond to the program order of instructions that access Device memory. A write is permitted to complete before it reaches the peripheral or memory component accessed by the write (bufferable).

System peripherals are almost always mapped as Device memory.

3) Normal memory
Normal memory is used to describe most parts of the memory system. Normal memory can be cacheable, bufferable. The core can prefetch or speculative access additional memory regions. Unaligned access can be performed. This region can also be either shareable or non-shareable.

These memory types can be summarized as table 5-1.

	Cacheable	Bufferable	Access order
Strongly-ordered memory	N	N	accesses in program order
Device memory	N	Y	accesses in program order
Normal memory	Y	Y	out-of order access

Table 5-1: Memory types model

The CPU private timer that we used in chapter four to implement the timer interrupt is an example of strongly-ordered memory. Since the access to this memory region isn't bufferable, next access to the registers of this private timer happens only when the previous access is completed.

The registers of UART that is mapped as memory-mapped IO is an example of Device memory. The characters displayed in the screen must be ordered (non-cacheable) but bufferable because UART registers are in Device memory region.

5.3.4 Memory Barriers

A memory barrier is an instruction that requires the core to apply an ordering constraint between memory operations that occur before and after the memory barrier instruction in the program. Such instructions can also be called memory fences in other architectures.

The term memory barrier can also be used to refer to a compiler mechanism that prevents the compiler from scheduling data access instructions across the barrier when performing optimizations. For example, in GCC, you can use the *inline assembler* memory clobber, to indicate that the instruction changes memory and therefore the optimizer cannot re-order memory accesses across the barrier. The syntax is as follows:
asm volatile("" ::: "memory");

ARM *RVCT* includes a similar intrinsic, called *__schedule_barrier().*

Here, however, we are looking at hardware memory barriers, provided through dedicated

ARM assembly language instructions. As we have seen, core optimizations such as caches, write buffers and out-of-order execution can result in memory operations occurring in an order different from that specified in the executing code. Normally, this re-ordering is invisible to the programmer. Application developers do not normally have to worry about memory barriers. However, there are cases where you might have to take care of such ordering issues, for example in device drivers or when you have multiple observers of the data that must be synchronized. The ARM architecture specifies memory barrier instructions, that enable you to force the core to wait for memory accesses to complete. These instructions are available in both ARM and Thumb code, in both user and privileged modes. In older versions of the architecture, these were performed using CP15 operations in ARM code only. Use of these is now deprecated, although preserved for compatibility.

Let's start by looking at the practical effect of these instructions in a single core system. This description is a simplified version of that given in the ARM Architecture Reference Manual, this section is intended to introduce the use of these instructions. The term explicit access is used to describe a data access resulting from a load or store instruction in the program. It does not include instruction fetches. To force an ordered access, following instructions can be used for a corresponding purpose [5].

1) Data Synchronization Barrier (DSB)
   This instruction forces the core to wait for all pending explicit data accesses to complete before any additional instruction stages can be executed. There is no effect on pre-fetching of instructions.

2) Data Memory Barrier (DMB)
   This instruction ensures that all memory accesses in program order before the barrier are observed in the system before any explicit memory accesses that appear in program order after the barrier. It does not affect the ordering of any other instructions executing on the core, or of instruction fetches.

3) Instruction Synchronization Barrier (ISB)
   This flushes the pipeline and prefetch buffer(s) in the core, so that all instructions following the ISB are fetched from cache or memory, after the instruction has completed. This ensures that the effects of context altering operations, for example, CP15 or *ASID* changes or TLB or branch predictor operations, executed before the ISB instruction are visible to any instructions fetched after the ISB. This does not, in itself, cause synchronization between data and instruction caches, but is required as a part of such an operation.

5.3.5 Address Space ID

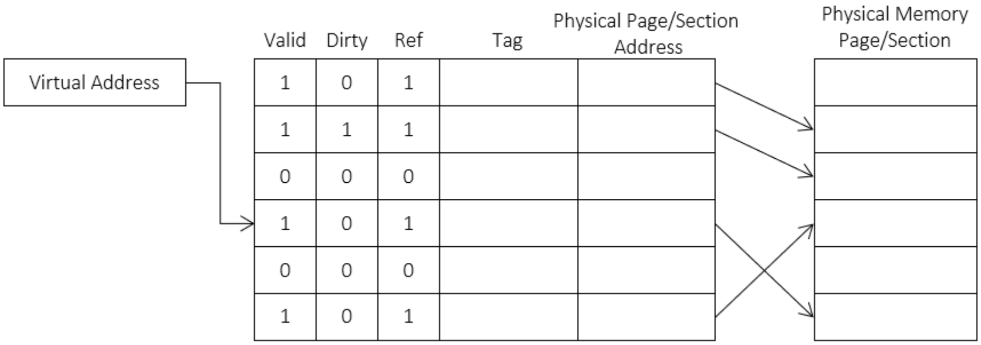
When we work on building the translation table in the future, we will see the bit called *nG (non-global)* in the configuration for section or page. If the nG bit is set for a particular page, the page is associated with a specific application. In this case, when the MMU performs a translation, it uses both the virtual address and an *ASID* value.

The ASID is a number assigned by the OS to each individual task. This value is in the range 0-255 and the value for the current task is written in the ASID register (accessed using CP15 c13). When the TLB is updated and the entry is marked as non-global, the ASID value will be stored in the TLB entry in addition to the normal translation information. Subsequent TLB look-ups will only match on that entry if the current ASID matches with the ASID that is stored in the entry. You can therefore have multiple valid TLB entries for a particular page (marked as non-global), but with different ASID values. This significantly reduces the software overhead of context switches, as it avoids the requirement to flush the on-chip TLBs. The ASID forms part of a larger (32-bit) process ID register that can be used in task-aware debugging.

5.3.6 Address Translation Sequence

Address translation is an expensive process. After virtual address space is turned on, every access to a memory needs a translation. The speed of external memory is much slower than CPU speed. As in figure 5-3, the translation table is placed in the external memory. To retrieve a data from the memory, there is two times of memory access needed, one for address translation table, the other for accessing the data region. The speed of single access of memory is tens of times slower than the core processor speed. So, LDR/STR instructions will stall the processor so long period. In this case, like memory data cache, the previous translation can be cached for the future access. The locality of accessing address can achieve much better performance by caching the previous translatioin. So, there is another type of cache called *Translation Lookaside Buffer (TLB)* cache. This TLB is a cache of translation table: cache of address recently used to avoid the access to translation table stored in the physical memory.

TLB is a lookup table looks like below figure 5-6. It looks like a normal cache: valid flag, dirty flag, reference count, Tag and data (address in this case). The MSB (Most Significatn Bits) bits in virtual address are compared to the tag of every entry in the table. If Small Page is used, the upper 20 bits (bit[31] ~ bit[10]) are used to this comparison.



Valid	Dirty	Ref	Tag	Physical Page/Section Address
1	0	1		
1	1	1		
0	0	0		
1	0	1		
0	0	0		
1	0	1		

Figure 5-6: TLB cache layout.

If TLB hits, we have a physical address directly from TLB and can build the physical address and access to that physical memory. If TLB misses, the MMU needs to traverse the translation table. This is very slow process because the translation table itself is stored into the physical memory. To address 4GB physical memory in Small Page format, the total physical memory needed to store the translation table entries can be calculated as:

1) Level 1 translation table has 4K entries, which needs (4Bytes * 4K) = 16KB memory.
2) Level 2 translatioin table has 4K * 256 entries, which needs (4Bytes * 4K * 256) = 4MB memory.

The total memory usage to store the translation table for Small Page format is 4MB + 16KB. This is not that much considering the total 4GB memory, it consumes about 0.1% of the total memory.

Each different translation table format has a different table walk scheme. Sectioin table walk and Small Page walk will be covered in chapter 5.5. In a high level, the virtual address translation step follows figure 5-7, TLB and translation table walk.

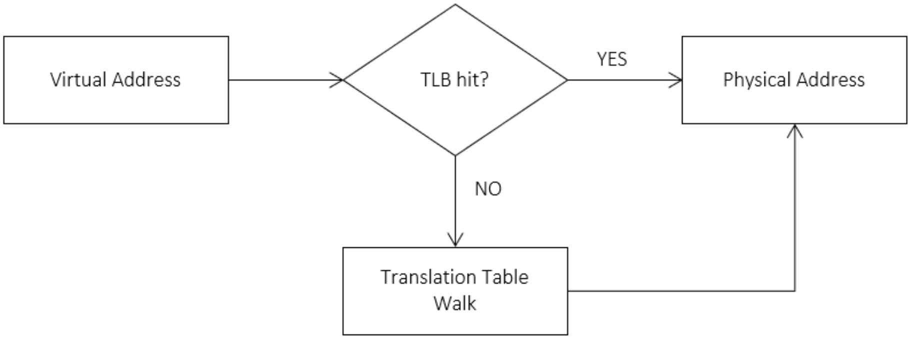


Figure 5-7: Address translation steps from virtual address to physical address

5.3.7 MMU Fault Checking

As described before, MMU's main functions are address translation and access permission checking. While translating the virtual address, the fault checking is also processed. The steps to faults in each step are depicted in figure 5-8 associated with translation table walk.

If *Alignment Fault* is enabled, MMU will generate alignment fault on the address which is not word-aligned. Alignment fault will not be generated for instruction fetch and any byte access. Alignment fault is enabled in SCTLR Register's *A bit, bit [1].*

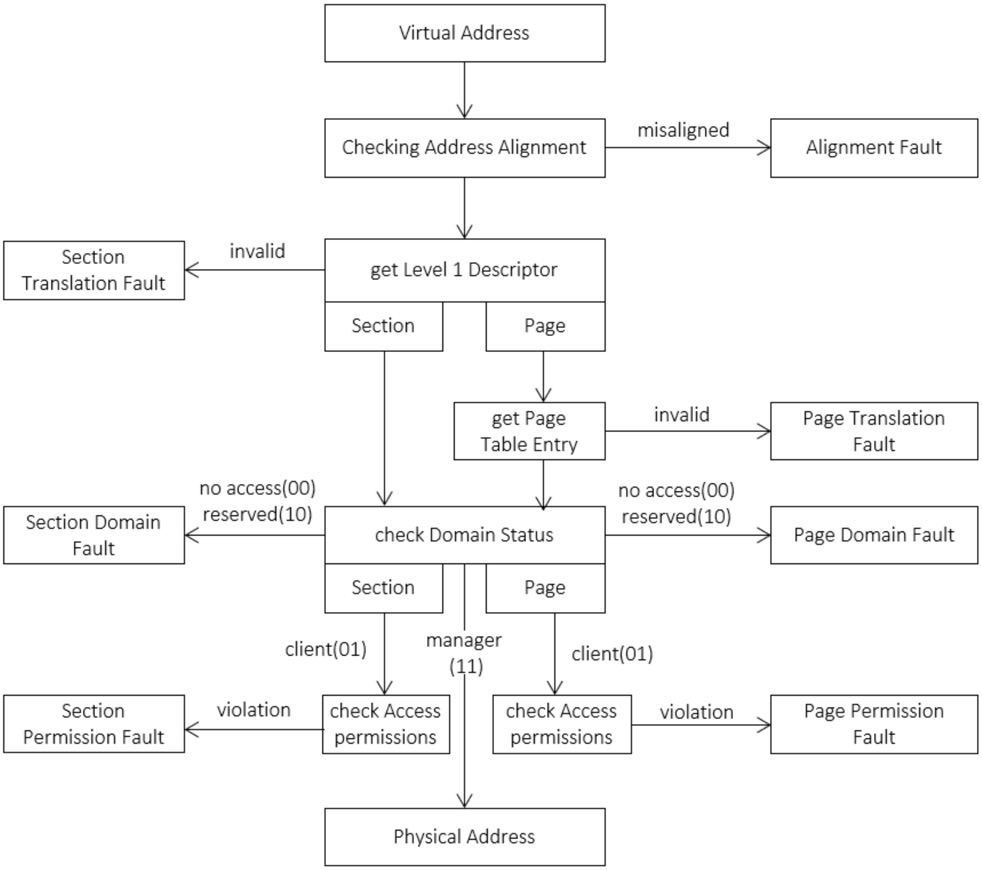


Figure 5-8: Virtual address translation sequence and fault checking

After passing the alignment check, MMU starts to translate the virtual address by walking the translation table entries. While translating the virutal address, if there is not a valid entry in the translation table, MMU generates a corresponding fault - *Page Translation Fault or Section Translation Fault*. Section translation fault occurs when there is not a valid entry in the level 1 translation table. Page translation fault occurs when there is no valid entry in the level 2 translation table. The *Demand Paging* feature in SLOS's memory manager uses these translation fault. SLOS memory manager doesn't set up the whole translation table entries for the heap region in the first. Rather, it creates the entry only when there is a need to access to a physical heap memory. We can call it a *lazy allocator*. Operating system developers seems so lazy people, aren't they? The deatiled description on this will be covered from chapter 5.7. After walking of these tables, MMU gets the physcial address of the base of Sectioin or Page.

Then, before accessing the physical memory, MMU checks the access permission of the domain after translation table walk. *Domain* is a 4-bit indexed memory region (up to 16 domains). The first level entry of translation table has its domain number (bit [8:5]) and the *Domain Access Control Register (DACR)* defines the access permission *(client, manager, no access)* for each of the 16 domains. The DACR register bits are like figure 5-9.

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

Figure 5-9: Domain Access Control Register (DACR) bit layout

The bit assignment for the access permission of n-th domain, Dn:

1) *2b00: No access*
   Any access to the domain generates a Domain Fault.

2) *2b01: Client*
   Accesses are checked against the permission bits in the translation tables.

3) 2b10: Reserved.

4) 2b11: Manager
   Accesses are not checked against the permission bits in the translation tables.

If domain check fails, MMU generates a permission fault. But if the domain property is set as a *Manager,* then the permisson check on the domain is bypassed. SLOS has only one domain and sets it as a *'Manager'* mode, which means SLOS doesn't check any access permissions.

In summary, the fault checking while MMU accesses to the physical memory from virtual address takes 3 steps: alignment check, page or section fault checking during address translation and access permission check on the domain before the physical memory access. After all these steps are passed, MMU can access the physical memory properly.

5.3.8 Translation Table Walk

After TLB miss happens, and after alignment checking passed, the MMU tries to traverse the translation tables. ARMv7 architecture defines two alternative translation table formats: *Short-descriptor format* and *Long-descriptor format.*

1) *Short-descriptor format*
   a) This is the only format supported on the implementation that do not include the *Large Physical Address Extension.*
   b) This uses 32-bit input addresses and 32-bit descriptor entries.
   c) Up to 2-levels of address lookup (Translation Table Walk).
   d) Support no access, client and manager access permissions for domains.

2) *Long-descriptor format*
   a) The Large Physical Address Extension adds support for this format.
   b) Up to 3-levels of address lookup.
   c) Input, Output addresses are up to 40bits.
   d) 4KB assignment granularity accross the entire PA range.
   e) No support for domain, all memory regions are treated as client domain.
   f) 64-bit table entries.

The Large Physcial Address Extension is optional. SLOS doesn't use it and supports only Short-descriptor format. All translation table formats after this assumes a Short-descriptor format.

For translation table walk, first, the MMU looks for the location of translation tables. ARM MMU stores the start address of translation table into its *Translation Table Base Register (TTBR) 0*, and 1. The TTBR register can be used either

1) TTBR0 can be configured to describe the translation of entire virtual address map, or
2) TTBR0 is configured to describe the lower part of virtual address map, TTBR1 is configured to describe upper part of virtual address map.

The first 3bits (*N, bit[2:0]*) in the *Translation Table Base Control Register (TTBCR)* determines whether the address map is separated into two parts, and where the separation occurs.

ARM MMU supports 4 kinds of translation table formats based on the memory block size it covers: *Supersection, Section, Large page table, and Small page table*. Supersection and Section formats have one level of translation table. Large page and small page formats have two levels of translation table. The translation tables in the second level are also called *page tables*.

1) *Supersection*

   Consists of 16MB blocks of memory. Support for Supersection is optional. The first-level translation table holds first level descriptors that contains the base address and translation properties for a Supersection.

2) *Section*

   Consists of 1MB blocks of memory. The first-level translation table holds first level descriptors that contains the base address and translation properties for a Section.

3) *Large Page*

   Consists of 64KB blocks of memory. The first-level translation table holds first level descriptors that contains the base address and translation properties and pointers to a second level table for a large page. Second-level tables hold second-level descriptors that contain the base address and translation properties for a large page.

4) *Small Page*

   Consists of 4KB blocks of memory. The first-level translation table holds first level descriptors that contains the base address and translation properties and pointers to a second level table for a small page. Second-level tables hold second-level descriptors that contain the base address and translation properties for a small page.

These translation tables are the keys to virtualize the physical memory address and must be programmed by the memory manager.

The smallest memory allocation size is also associated to the memory block size that is addressed by one entry in translation table. For example, the smallest memory size that Supersection can allocate is 16MB, and Small Page has 4KB for its smallest memory allocation. This brings about the memory fragmentations: there are two types of memory fragmentations, *Internal Memory Fragmentation* and External Memory Fragmentation*.* Internal fragementation, as its name suggests, happens internal of the memory block. Even though the application requests 1MB of memory, the memory manager still allocates 16MB in Supersection. Then, the rest 15MB is unused. This is why it is called an internal fragmentation. The external fragmentation also occurs, for example, when memory manager has 17MB free in Supersection. This means there is no way to allocate the rest 1MB by the memory manager.

If memory block size is bigger, then there are more chances of fragmentations in the memory. So, the Supersection suffers from larger fragmentation than Small Page. But if the block size is bigger, there is more chance to hit the TLB cache, which results in better access speed. So, there are upside and downside in each format. SLOS's memory management uses Small Page translation tables.

5.3.8.1 Section Table Walk

ARM MMU has two types Sections as we discussed before. In this chpater, we will cover the Section translation table format and how the translation table walk happens. The *LSB (Least Significant Bits)* in the virtual address are used to designate the offset in each memory block. That means the bit[23:0] of a virtual address defines the offset in the 16MB Supersection block, bits[19:0] of a virtual address defines the offset in the 1MB Section block, bits[15:0] of a virtual address defines the offset in 64KB the Large Page, and bits[11:0] defines the offset in the 4KB Small Page.

The Section translation table walk is depicted in figure 5-10. The virtual address in Section is splitted into table index part, bit[31:20] and section offset part, bit[19:0]. The table index part is used for the entry index of first level translation table. To access the translation table, MMU first reads the TTBR register to get the translation table base address, then it merges it with table index from virtual address bit[31:20]. This makes the address for the first translaton table entry of the virtual address. Now, MMU reads the first level descriptor from physical memory. The first level descriptor is composed of the section base address bits (bit[31:20]), and of the property bits (bit[19:0]). The property bit of Section format looks like figure 5-9. The first level Section descriptor must have property bits. Let's look into this more detail.

1) *Descriptor Format, bit[1:0]*
   These bits are used to set the descriptor format. 2'b00 is for invalid, which results in Section Translation Fault. 2'b01is set for Page Table, which means the descriptor gives the address of second-level translation table. 2'b10 is set for Section or Supersection. bit[18] value 0 is for Section, and bit[18] value 1 is for Supersection.

2) *TEX[2:0], C, B*
   These bits are used for memory region attributes. It controls the memory type, accesses to the caches, shearable and so on. There are three types of memory defined in the ARM architecture. All regions of memory are configured as one of following three types. We already discussed this in chapter 5.3.3 and here is a short notes.

   Strongly-ordered Region:
   This memory region doesn't allow any cache, buffer and out-of-order access to the memory. ARM defines some memory area must be a strongly-ordered region such as CPU private registers. All Strongly-ordered accesses are assumed to be shared.

   Device Memory Region:
   This memory type is used for memory-mapped peripherals. The I/O device memory (device registers in MMIO) needs a non-cacheable property. It can still be bufferable.

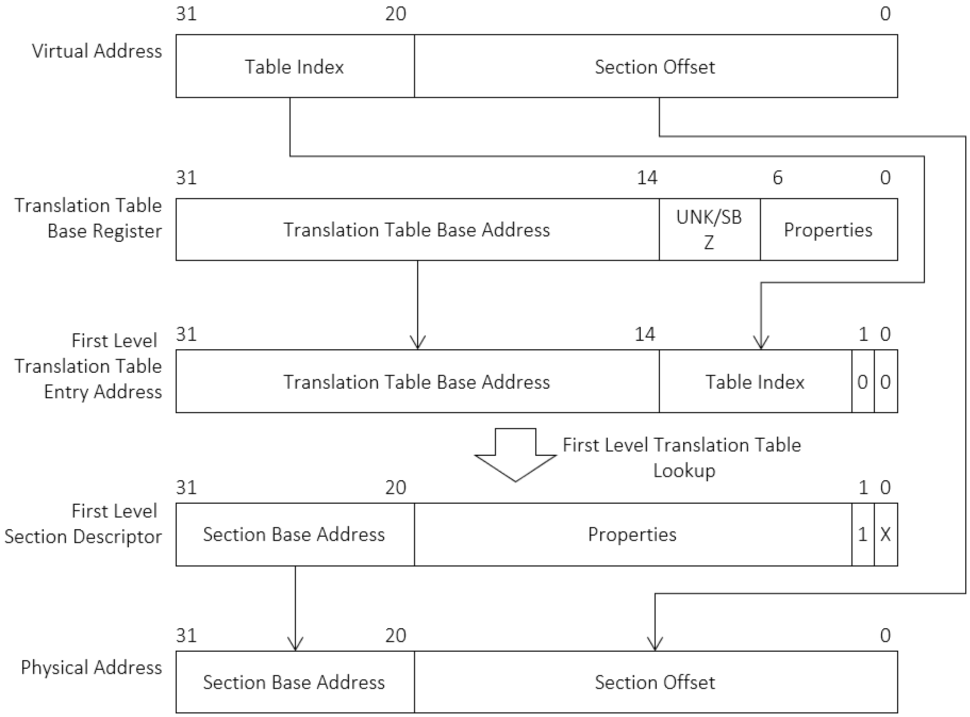

31
20
0
Virtual Address
Table Index
Section Offset
31
14
6
0
Translation Table Base Register
Translation Table Base Address
UNK/SBZ
Properties
First Level Translation Table Entry Address
31
14
1 0
Translation Table Base Address
Table Index
0
0
First Level Translation Table Lookup
31
20
1 0
First Level Section Descriptor
Section Base Address
Properties
1
X
31
20
0
Physical Address
Section Base Address
Section Offset

Figure 5-10: Section translation table walk

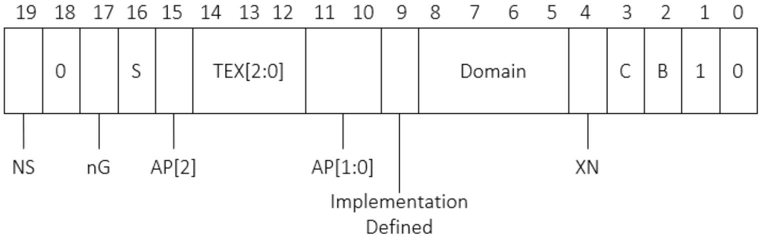


Figure 5-11: Property bits of first level transltation table in Section table walk

All memory access to Device Memory occur in program order.

Normal Memory Region:
This region allows cacheable, bufferable and even out-of-order access. Normal Memory region also can be either shareable or non-shareable between multiple cores.

Below table 5-2 shows how the TEX, C, B bits in translation table entry are used to

TEX	C	B	Description	Memory Type
000	0	0	Strongly-ordered	Strongly-ordered
000	0	1	Shareable device	Device
000	1	0	Outer and Inner write-through, no allocate on write	Normal
000	1	1	Outer and Inner write-back, no allocate on write	
001	0	0	Outer and Inner non-cacheable	Normal
001	-	-	Reserved	-
010	0	0	Non-shareable device	Device
010	-	-		
011	-	-		

Table 5-2: Properties of TEX, C, B bits

set the memory type and cache policies used.

3) *XN bit, Execution Never* bit.
This bit determines whether the processor can execute software from the addressed region.

4) *Domain*
These bits are domain field. A domain is a collection of memory regions. The Short-descriptor translation table has 16 domains and requires the software that defines a translation table to assign each virtual memory region to a domain. Only the first level translation table defines domain and all second level translation table inherits the domain from the domain in the first level translation table. Each domain has its own access permissions - *No access, Client, Manager.*

5) AP[2], AP[1:0], Access Permission bits
These bits control the access to the corresponding memory regions. There are two options for defining the access permissions in Short-descriptor translation table format.
a) All three bits are used. AP[2:0] define the access permissions.
b) AP[2:1] define the access permissions and AP[0] can be used as an Access flag.
Since SLOS uses the option a), let's find out the permissions by the AP[2:0] bits. For other permission types, refer the ARM reference document [2] as always.

AP[2]	AP[1:0]	PL1 access	Unprivilege access	Description
0	00	No access	No access	All accesses generate permission faults

	01	Read/Write	No access	Access only at PL1
	10	Read/Write	Read-only	Writes at PL0 generate permission faults
	11	Read/Write	Read/Write	Full access
1	00	-	-	Reserved
	01	Read-only	No access	Read-only at PL1
	10	Read-only	Read-only	Deprecated
	11	Read-only	Read-only	Read-only at any privilege level

Table 5-3: ARM MMU access permission

6) *S, Sharable bit*
   This bit determines whether the addressed region is *Sharable memory.*

7) *nG, not Global bit*
   This bit determines global or non-global memory region.
   a) 0: The translation is global, meaning the memory region is available for all processes.
   b) 1: The translation is non-global, or process-specific, meaning it relates to current *Address Space IDentifier (ASID)*

8) *NS bit, Non-Secure* bit
   If *Security Extension* is supported, this bit specifies whether the translated *PA* is in the Secure or Non-secure address map. Secure mode and Non-secure mode have their own translation tables which means they have their own TTBR0, TTBR1, TTBCR registers for the banked translation tables.

9) Bit[18]
   When bits[1:0] indicates a Section or a Supersection descriptor,
   a) 0: Descriptor is for a Section.
   b) 1: Descriptor is for a Supersection.
   Since we are using a Section descriptor in this chapter, this bit must be set 0.

Some of these bits still appear in Small Page Tables, but some doesn't. The Sharable bit doesn't appear in level 1 descriptor in Small Page Table.

After translation table walk done, the MMU gets the base address of a specific Section. The final address is calculated by adding the offset from the base of the Section as figure 5-12. Section table has Section (1MB) granularity in the physical memory.

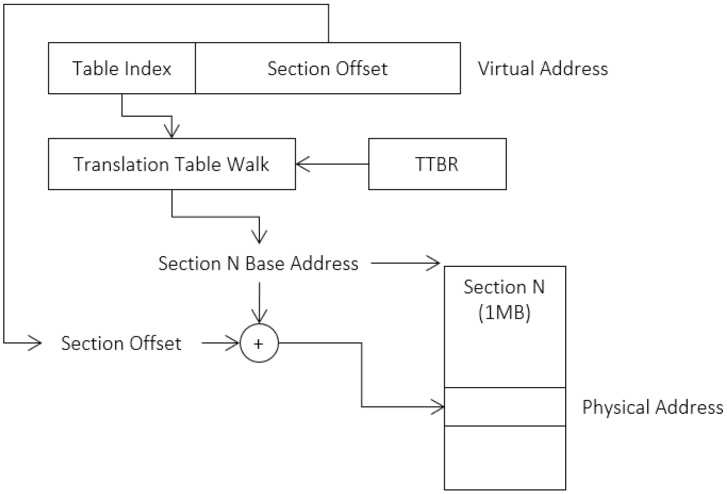


Figure 5-12: Final address generation after Section translation table walk

5.3.8.2 Small Page Table Walk

ARM supports two types of pages: Large Page and Samll Page. Large Page has 64KB physical memory granularity and Small Page has 4KB physical memory granularity. Small Page, 4KB physical memory granularity, means it has 4KB (12bit) offset range in its virtual address. Both small page table and large page table formats have two-level translation table walk.

Small Page walk traverses the translation table as in figure 5-13. The virtual address is splitted into three parts: level 1 table index, level 2 table index and page offset. Each has 12bit, 8bit, 12bit width. The level 1 table index is combined with the translation table base address in TTBR register. This is the address of first level translation table entry. The level 1 table index is 12bit wide, that means there are 4096 entries in the level 1 translation table. In two level translation table, the descriptor of level 1 table has an address of level 2 table entry. Since all L1 entries are an address of L2 page table base address, the last two bits are 2'b00. Each entry size is 4 byte (32bit address), the size of level 1 translation table must be 4K entries * 4Bytes = 16KB.

The left-most 22bits (bit[31:10]) in level 1 translation table entry is the base address of level 2 translation table. This base address is combined again with the L2 table index in the virtual address. After the L2 table base address bits, the eight property bits (bit[9:2])follows. The rest two bits (bit[1:0]) has 2'b01 for page (Small or Large) translation table. The property bits of first translation table of Page tabel look like as figure 5-14. All of them are already covered in previous chapter. The *C, B, AP, TEX, nG, S bits* are moved to the second level translatioin table descriptor in this case.

L2 table index is 8bit wide, which means each L2 table has 256 entries. The page table base address in L1 descriptor is combined with the level 2 table index in the virtual address in order to generate the entry address to the entry of L2 translation table. The descriptor in second translation table has a page base address and the properties of that page. Since Small Page size is 4KB, the page base address has 20bits width. The property bits in the second translation table is described in figure 5-15. All of those property bits are already covered in Section table property. The last two bits (bit[1:0]) are set as 2'b10 for Small Page, and 2'b01 for Large Page. The most property bits are determined in the second translation table that is the last entry before getting the physical address.

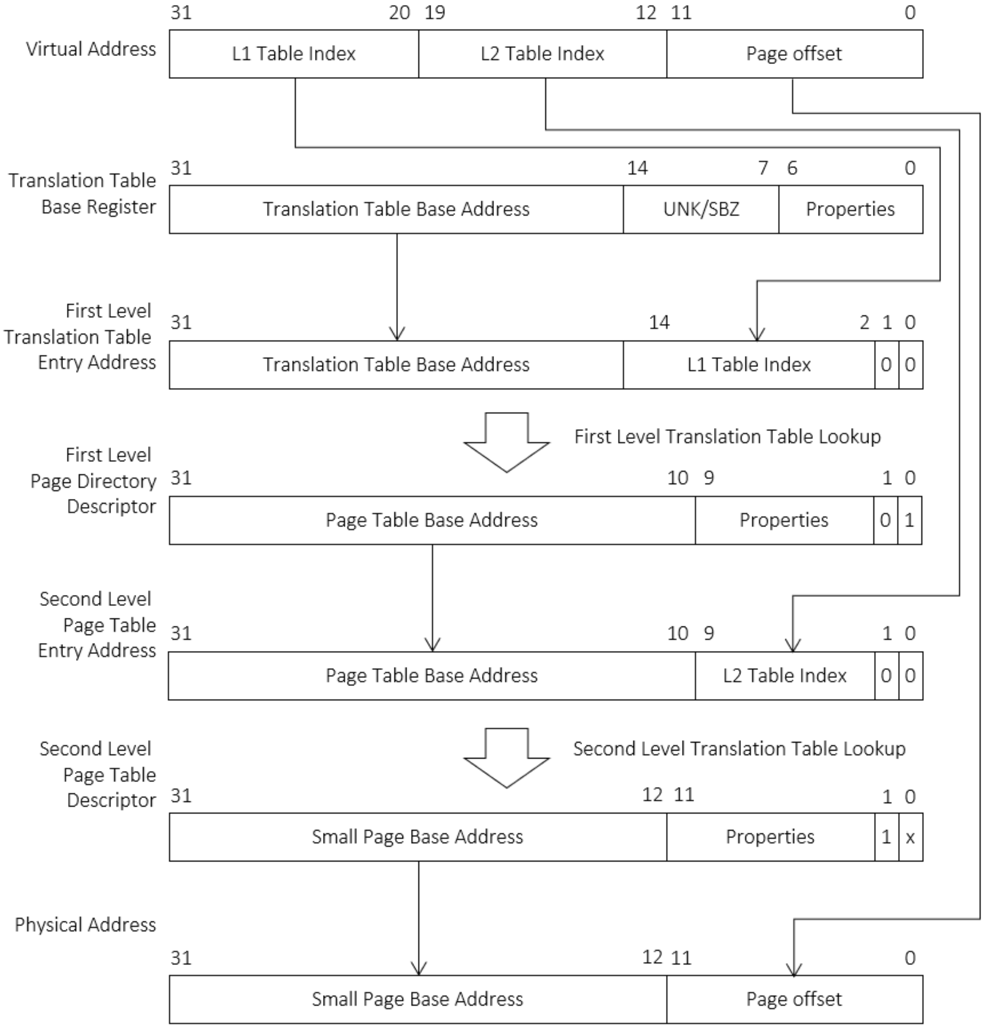

31
20
19
12
11
0
Virtual Address
L1 Table Index
L2 Table Index
Page offset
Translation Table Base Register
31
14
7
6
0
Translation Table Base Address
UNK/SBZ
Properties
First Level Translation Table Entry Address
31
14
2
1
0
Translation Table Base Address
L1 Table Index
0
0
First Level Translation Table Lookup
First Level Page Directory Descriptor
31
10
9
1
0
Page Table Base Address
Properties
0
1
Second Level Page Table Entry Address
31
10
9
1
0
Page Table Base Address
L2 Table Index
0
0
Second Level Translation Table Lookup
Second Level Page Table Descriptor
31
12
11
1
0
Small Page Base Address
Properties
1
x
Physical Address
31
12
11
0
Small Page Base Address
Page offset

Figure 5-13: Small Page translation table walk

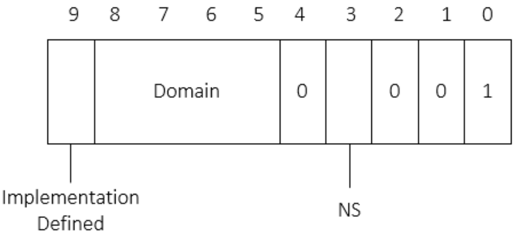


Figure 5-14: Property bits of first level transltation table of Page table walk

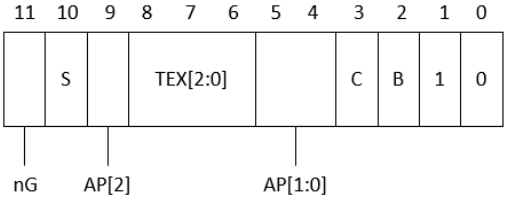


Figure 5-15: Property bits of second level transltation table of Page table walk

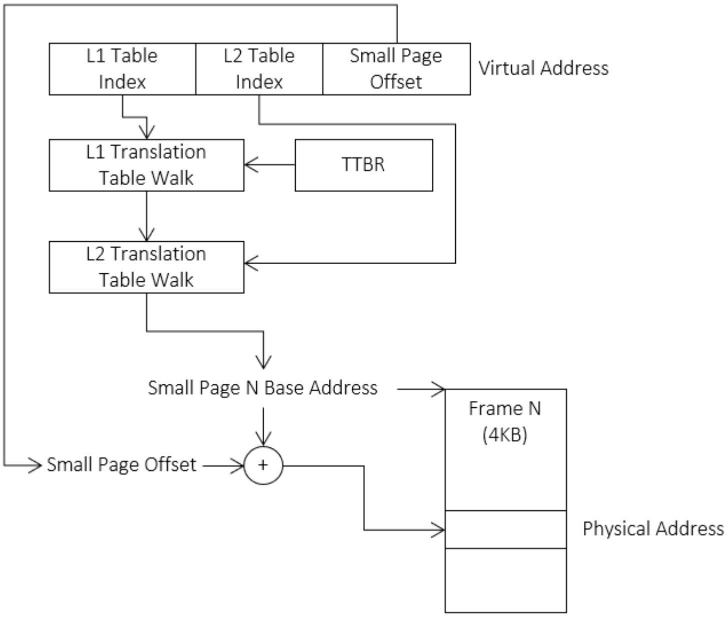


Figure 5-16: Final address generation after two level page translation table walk

The MMU gets the base address of a specific Small Page from the second level translation table. This page's base address is combined with the page offset (12bits) in the original input of virtual address. The final address is obtained as figure 5-16. The whole process is pretty similar with Section table walk in figure 5-12 except Small Page walk has two level translation tables. The content of level 1 translation table descriptor contains the base entry of level 2 translation table. We are going to use *Frame* rather than *Page* for 4KB physical memory block. SLOS will implement the frame manager for managing the physical memory block later. Small Page table format has a 4KB granularity in the physical memory which is equal to the size of memory frame.

5.3.9 Memory Access - Altogether

We are digging into the address translation from virtual address to physical address, mostly focusing on the MMU and its translation table walk. But besides address translation, in order to access the data in memory, there is another important hardware module: Cache. Cache is a small, fast memory which is tightly bonded with the processor. Actually, it is cache memory that the core processor interacts with for *load and store* its data or instruction code. External RAM memory is just another next level of memory storage like hard disk storage. There are many different schemes to maintain a consistency in caches based on cache type, processors (MP vs UP) and so on. We will not cover these topics in this book since there are already many good books, articles on the web and especially we don't need to know about them in implementing our simple and light OS. They are mostly handled by hardware. Nonetheless, we are going to enable/disable cache, invalidate, flush cache functions later in custom DMA implementation in chapter 7 and that can be done with simple ARM ISA instructions.

Even the hardware manages cache, it is very good to understand the full path to memory from CPU processor. Let's draw a picture of that and keep it in mind. The full path to memory access is depicted in figure 5-17. Figure 5-17 is a detailed version of figure 5-3. In the higher view in figure 5-3, there are only processor, MMU, cache and external memory. Figure 5-17 is a system-wide scope to access to a data in memory with more details. In order to access some specific location, we need its location. So, it first checks the TLB cache to figure out whether the mapping of virtual address to physical address is hit or not. If TLB hits, it gets the page physical address from the TLB and can build the complete physical address combining with the page offset value. If TLB misses, the MMU walks the translation tables to get the page physical address as we covered before.

Then, after build the complete physical address, MMU interprets it in a different way to access the cache memory. It splits the physical address with *Tag, Index and Byte Offset.* Index is used to the entry index of a *cache line* and tag is used to determine whether current cache line hits or not. If it hits, read the data from the cache line by using the byte offset. If cache misses, MMU accesses the outer physical memory to read or write the data from memory. This seems a long path and whenever caches (TLB or Cache) miss, the processor should stall until the physical outer memory access is completed. But still, in many cases, caching data by taking advantage of access data locality is a better way for performance.

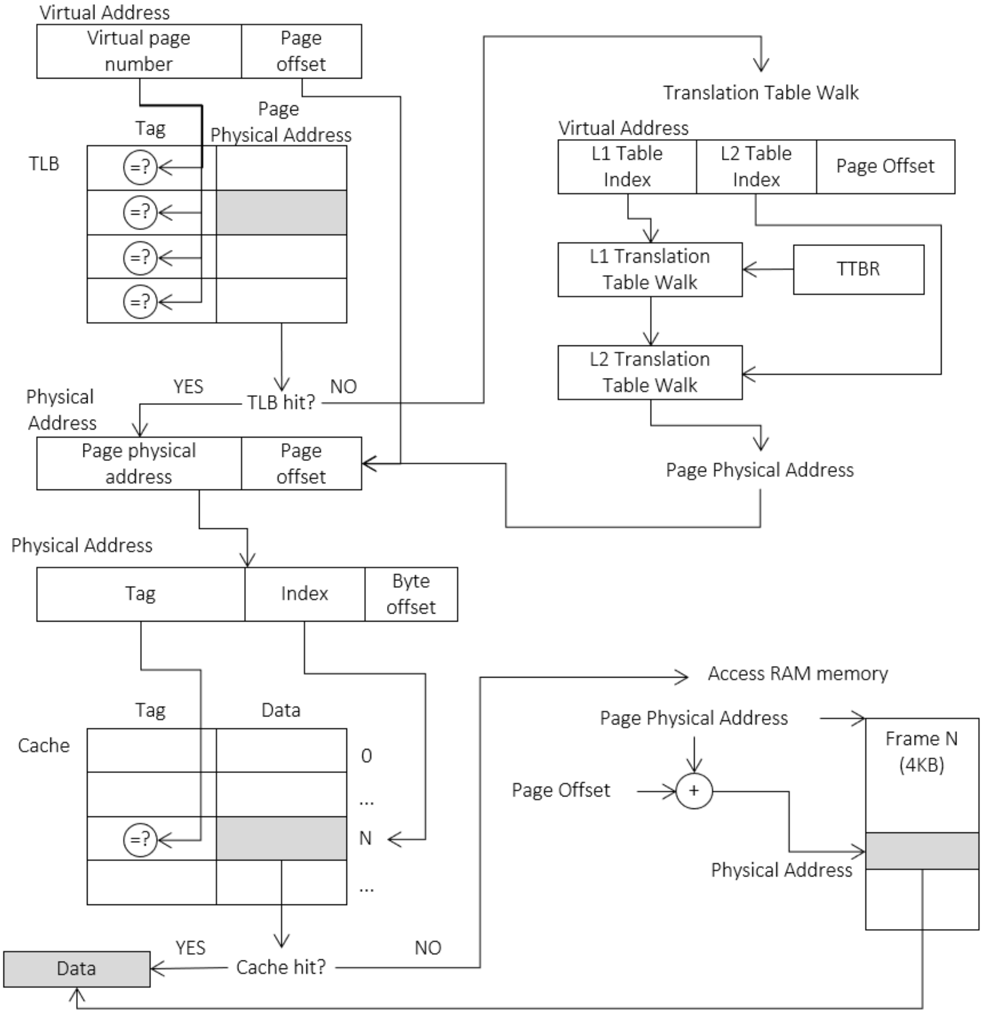


Figure 5-17: Full path to physical memory access from virtual addresss

5.4 SLOS Virtual Memory

We've learned the MMU and its basic operations so far. Now, we are going to look into our real interests - how the memory management of SLOS can be implemented. First thing we need to know is that how we rebuild the memory layout by using the virtual address space. Let's dive into the real world now. The SLOS source files for memory management implementation can be obtained by running below command.

git checkout SLOS_CH5

5.4.1 Linker Script Update

As described in chapter 3.3.1, linker script has two types of address; *VMA (Virtual Memory Address) and LMA (Load Memory Address)*. The chapter 4, the process management has the same VMA and LMA. But in chapter 5, the memory management makes them different addresses. We already covered the MMU's key role that is the abstraction of physical memory and virtualize the physical address. So, in order to access the physical memory, we need to interpret what the virtual address means.

In SLOS, the kernel *text, data* section physical location is not changed: it is still located at the address from 0x0010_0000. But they will be loaded into 0xC010_0000 address when kernel starts up. In other words, the memory loaded address (LMA) is still 0x0010_0000 but the virtual memory address (VMA) that is logically used in the processor is changed to 0xC010_0000. Remember that the virtual address is same as the logical address in this case. For this, the linker script of process management in chapter 3.3.2 is changed as below.

```
OUTPUT_ARCH(arm)
ENTRY(ssbl)
KERNEL_HEAP_START = 0xC4000000;
KERNEL_HEAP_SIZE = 0x4000000;

SECTIONS
{
    . = 0x100000;
    .ssbl : {
        *(SSBL);
        *(PGT_INIT);
    }
    . = 0xC0101000;
    .boot : AT(ADDR(.boot) - 0xC0000000) {
        *(EXCEPTIONS);
        *(.text);
    }
    .data : AT(ADDR(.data) - 0xC0000000) {
        *(.data)
    }
    .bss : AT(ADDR(.bss) - 0xC0000000) {
        *(.bss)
    }
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    }
```

The basic layout sequence is still same, but there are two big changes in this linker script. One is that there is another section named *.ssbl.* This section is called a *secondary bootloader (SSBL)* which is setting up the virtual address translation tables for MMU to walk. It is named as a bootloader but it is not a separate file, it is just another section of a kernel.elf binary. The .ssbl section, as the name imply, must be run before the kernel reset handler. The entry point of the kernel is now changed to the start address of this section. The line 2 sets this new entry address and line 8 ~ 12 adds a new .ssbl section into the output section. Since the codes in ssbl secion run in MMU disabled, the LMA and VMA are same and the start address is still 0x0010_0000.

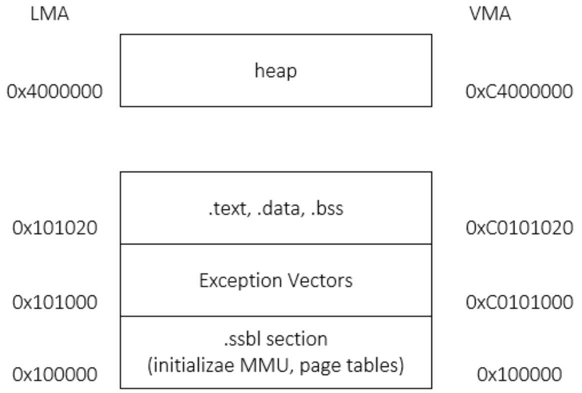


Figure 5-18: Layout of kernel sections having different LMA and VMA

The second change is that the kernel's real start address starts from 0xC010_1000. Notice that the line 13 sets the *current location counter* value as 0xC010_1000 and kernel reset_handler starts from that address. This address is a VMA. We don't want to change the physical address that the output section is loaded. Since the LMA still starts from 0x0010_0000, the *'AT'* keyword is used to specify the load address of the section. With *AT* keyword, we can set the different LMA and VMA in the linker script. The LMA can be calculated by subtracting the current LMA with the offset value 0xC000_0000. So, every section after .boot section has distinct LMA and VMA, sets its LMA by using *AT* keyword after the section name. All sections of kernel are still loaded from 0x0010_0000 physically but logically it starts from 0xC010_1000. Remember that if we enable MMU, the CPU's logical address is same as the virtual address. We already covered that before. Below figure 5-18 describes the new layout 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, low memory address. So, let's put the SLOS kernel into high address range starting from 0xC010_1000.

5.4.2 SLOS Virtual Memory Map

After enabling MMU, SLOS starts to use virtual memory address. Then, the memory map of SLOS in chapter 4 must be changed considering the virtual address space. SLOS has a virtual memory map as depicted in figure 5-19. It has a 4GB virtual memory address range. 0x0000_0000 ~ 0xBFFFFFFF, 0xE0000000 ~ 0xFFFFFFFF regions are *directly mapped address.* Directly mapped address is an address whose virtual address is same as its physical address. After translation table walk, the physical address value is same as the virtual address in those address ranges. The directly mapped address is necessary for some address regions whose address ranges are already allocated for the system. AXI interface address region, IO peripheral region, SLCR region, PS subsystem region, CPU private registers region are all located at this directly mapped address region. The virtual address 0xC000_0000 ~ 0xDFFF_FFFF region is reserved for SLOS's kernel and is remapped to the physical address of 0x0000_0000 ~ 0x1FFFFFFF. This is because the kernel virtual address is started from 0xC0000000 region and this address region should be translated to the 0x0000_0000 ~ 0x2000_0000 physical address. So, the virtual address translation for 0x0000_0000 ~ 0x1FFFFFFF and 0xC0000000 ~ 0xDFFFFFFF are exactly same, resulting in the 0x0000_0000 ~ 0x1FFF_FFFF physical address. Normally, the lower address region starting from 0x0000_0000 in virtual address space is assigned to the custom applications and higher address region is assigned to the kernel.

SLOS memory manager implements a Small Page translation table walk which has 4KB size memory granularity. Since each entry in *page directory table* (Level 1 translation table) has 12 bits as described in figure 5-13 and it can cover 1MB physical address region, there are 4K entries in page directory table to address the total 4GB virtual address space. Each entry is 4B size and storing 4K entry table needs 16KB space. As we discussed before, these translation table also placed at some area of outer RAM memory. SLOS memory manager puts the page directories at 0x0000_0000 ~ 0x0000_4000 region. Since the page directory size is 16KB, the base address of page directory should be 16KB aligned. Those are equivalent

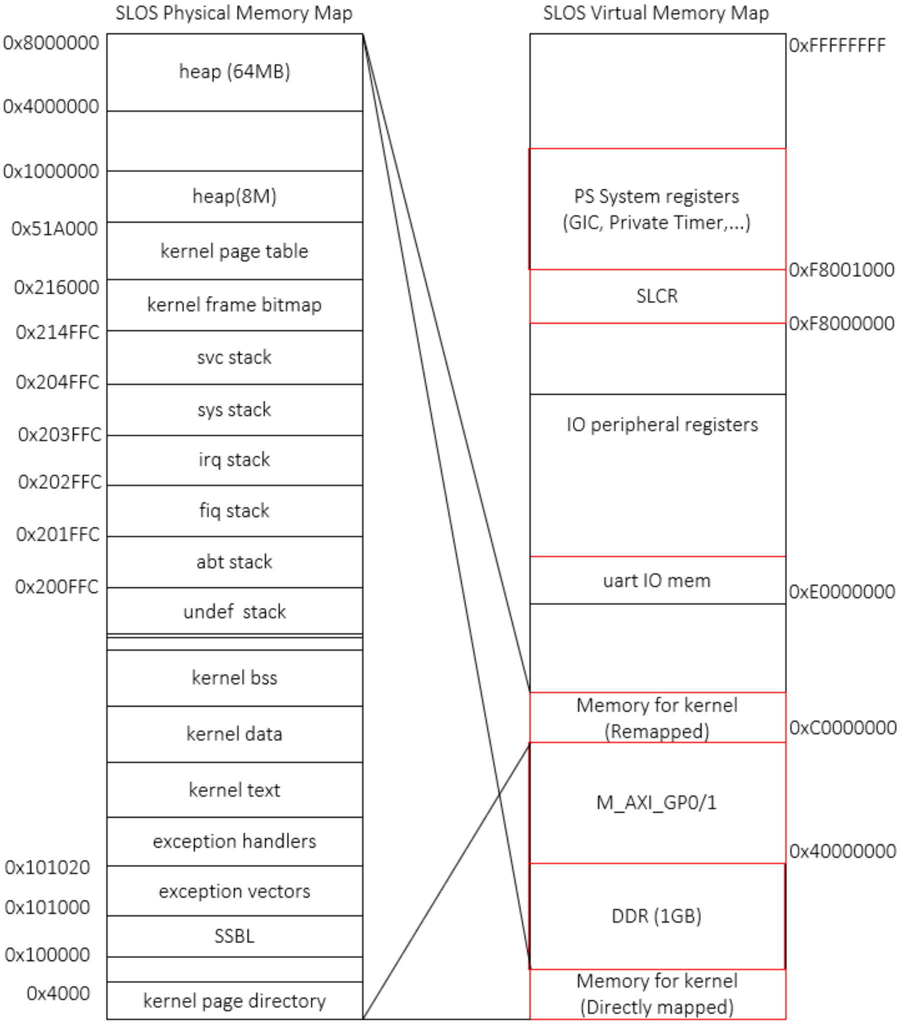


Figure 5-19: SLOS virtual address memory map

to the 4 *memory frames* having 4K entries for addressing the 4GB memory region. The physical 4KB memory region is called *memory frame* and page is a term used for MMU's translation table which virtualizes the memory frames.

Kernel start address is still 0x0010_0000 and up to 0x0021_5000 as in figure 4-2. This region is not changed from chapter 4. Right on top of kernel stack, there is a 4KB memory frame bitmap for managing the kernel frames. Kernel manages its physical memory resource using 4KB frames and this bitmap shows the allocation of physical memory frame. If a certain bit is set, then the associated memory frame is already allocated. We know that the output result of translation table walk is the base address of this 4KB memory frame. So, the SLOS memory manager splits the whole memory with the 4KB frames and maintain a bitmap to keep track of the allocation status. One bit of the memory frame bitmap covers 4KB physical memory frame. One frame bit map which is 4KB can address 128MB (4K * 8 * 4KB = 128MB) physical memory region and this is enough for SLOS. The frame bitmap will be mostly used for heap allocation. We will cover this later.

Page table of secondary translation is started after the frame bitmap and is ranged from 0x21600 to 0x51A000. This table has 256 * 4K entries. In this chapter, SLOS heap is 64MB size and starts from 0x4000000 to 0x8000000. All these regions are placed into DDR memory region.

As in figure 5-19, the physical memory for SLOS kernel is remapped into two places in the virtual address space. One is directly mapped region which has the same virtual address and physical address, and the other is remapped through the Small Page translation table.

Except the SLOS kernel virtual address, all the other memory region such as M_AXI_GP0/1, IO peripheral registers, SLCR are directly mapped. So, we can access those registers with the same address defined in the zynq7000 TRM document.

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 means 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 address translation tables are initialized early before reset handler of kernel. The reset handler is not placed in the start address now. When *FSBL* bootloader jumps to the address 0x0010_0000, the boot sequence meets the entry of secondary bootloader which is put into *.ssbl* section. This secondary bootloader section is placed the start address by the linker script (remember the two roles of linker script). The secondary bootloader is composed of MMU initialization routines and page table initialization routines.

Figure 5-20 describes the memory manager initialization flow in the secondary bootloader. For the initialization of MMU, caches, you can refer the ARM Architecture Reference Manual [2] but it has quite too much information for our simple OS. There is a summary of CP15 register in chapter B3.7.2 of the ARM reference manual document. The file *kernel/exception/init_mm.S* manipulates the CP15 registers to set the MMU. Before setting up the translation table, it first invalidates the TLB, I cache, D cache. Changing the content of TLB needs an invalidation of TLB. After invalidating a TLB, I cache, D cache, it calls the page table setup routine which is in *kernel/core/mm.c*. The translation table setup routine fills up the table content necessary for the steps in figure 5-13. While filling up the entries of translation table, it also should set the correct properties.

The page translation table is created in init_pgt() function in mm.c file. Since this function should be built into the .ssbl section, the *'section'* attribute keyword is used in the function declaration. This function is declared as

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

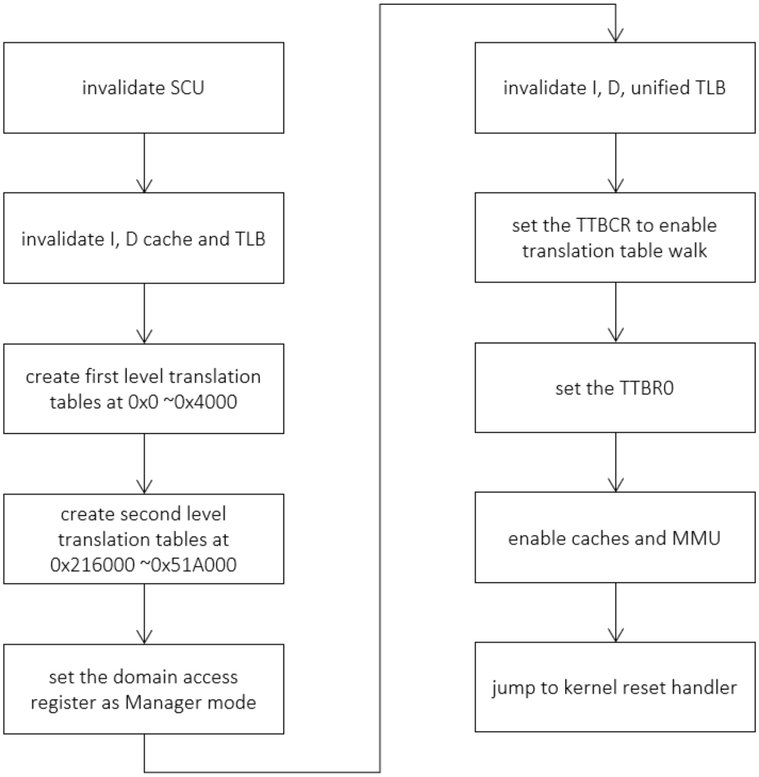

invalidate SCU
invalidate I, D cache and TLB
create first level translation tables at 0x0 ~0x4000
create second level translation tables at 0x216000 ~0x51A000
set the domain access register as Manager mode
invalidate I, D, unified TLB
set the TTBCR to enable translation table walk
set the TTBR0
enable caches and MMU
jump to kernel reset handler

Figure 5-20: Initialization of MMU and page table in a secondary bootloader

This puts the init_pgt() function into the section named as *'PGT_INIT'* and the linker script line 9 ~11 in chapter 5.4.1 will place the PGT_INIT input section together with *SSBL* input section to the secondary bootloader's *.ssbl* output section.

After finishing setup of translation table, the secondary bootloader code in init_mm.S runs again. It finishes the rest part of MMU setup. Those are invalidation TLB, caches, setup TTBCR, TTBR registers, enabling caches and MMU. Then, it finally jumps to the SLOS reset handler. Since the MMU is turned on, all logical address is same as virtual address now and needs to be translated through the translation table walk.

5.5.2 Initialization of First Level Translation Table

The init_pgt() function first initializes the first level 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 which is 4 physical memory frames. Each entry of this table covers 1MB memory region and 4K entry can address 4GB memory in total. The first entry of this table points the base address of second level translation table. Since each entry of the first translation table points to the second level of translation table which has 256 entries, the address of entry in first translation table is increased by 0x400 which is 1KB size (256 entries * 4B). So, the upper 22bits are used to point to the base address of second level translation table and the lower 10bits are used for setting the property of memory region. Check this in figure 5-13, Samll Page translation table walk. The init_pgt() function sets this property bits (bit[9:0]) as 0x1E1. 0x1E1 means:

1) *Bit[1:0], descriptor type*
   a) 2'b00: Invalid. The associated VA is unmapped. Any attempt to access that address will generate a section translation fault.
   b) 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.
   c) 2'b10: Section or Supersection. The descriptor gives the base address of the Section or Supersection. Bit[18] determines whether the entry describes a Section or a Supersection.
   d) 2'b11: reserved.

   We set this value 2'b01 for small page regioins or set this bit 2'b00 for its heap region.

2) *Bit[2], PXN (Privilege eXecution Never)*

   SLOS doesn't support this feature. This bit must be 1'b0.

3) *Bit[3], NS (Non-Secure)*

   This bit is 1'b0 because SLOS supports only *Secure Mode* operation.

4) *Bit[4], SBZ (Should Be Zero)*

   This bit should be 1'b0.

5) *Bit[8:5], Domain*

   This bit is set as 4'b1111 for setting domain as 0xF. SLOS has only one domain haveing domain number 0xF (4'b1111).

6) *Bit[9], Implementation Defined*

   This bit is set as 1'b0. don't care.

Normally, translation table entries for all regions of memory that are not peripheral I/O devices must be marked as L1 Cacheable and (by default) set to read-allocate, write-back cache policy.

SLOS's memory manager implements a 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 section translation fault.

The first level translation table doesn't have the entries for the heap region which is ranged from 0xC4000000 to 0xC8000000. The init_pgt() funcction doesn't setup the entries for the heap, i.e. the values for translation table entry for the heap is 0. If there is a trial to access to the heap region such as kmalloc() to allocate a heap memory, then it will generate a section translation fault. SLOS's memory manager handles this fault by demanding new pages. The *Demand Paging* feature will be covered in chapter 5.7. Anyway, first level translation table entry's bit[1:0] can be either 2'b00 for the heap region or 2'b01 for all other regions.

Value of Bit[4:2] for *PXN, NS and SBZ* is 3'b000. Bit[2] is PXN(Privilege eXecution Never) and it should be 0. SLOS doesn't support any kinds of *XN, PXN* features. Bit[3] is to set either secure mode translation table walk or non-secure mode translation table walk and set it 1'b0. This means SLOS supports secure mode translation table walk. Based on secure mode, the translation tables are banked. Secure mode walks its own translation table and non-secure mode walks its own translation tables. Bit[4] should be 1'b0.

Bit[8:5] designates the domain of memory. The init_pgt() function sets the Bit[8:5] for domain as 4'b1111 which is for domain 15 for the whole memory regions. 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 described in figure 5-9. Permission values are also discussed but let's write it down here again.

1) 2'b00, *No access*
   Any access to the domain generates a Domain Fault

2) 2'b01, *Client*
   Accesses are checked against the permission bits in the translation tables.

3) 2'b10: *Reserved*

4) 2'b11: *Manager*
   Accesses are not checked against the permission bits in the translation tables.

Since we don't want any access permission check to the access of domain, all domains' access permission value is 2'b11 as a manager permission. The DACR register in CP15 can be set as below ARM ISA instruction. This sets the DACR register 0xFFFFFFFF, which sets all domains as a manager mode.

```
ldr     r0, =0xFFFFFFFF
```

```
        mcr        p15, 0, r0, c3, c0, 0
```

Bit[9] is don’t-care bit and is set as 1’b0.

Then, the property field of first translation table entry has value 0x1E1. The left 22bits of first level translation table is the start address of the 256 entries of second level translation table. The first translation table entry is set as below pseudo code.

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

KERN_PGD_START_BASE in line 1 is defined as 0x0000_0000 in SLOS. The first level translation table are placed at the very start address of address space. The *KERN_PGT_START_BASE* is the base address of second level translation table and is defined as 0x0021_A000 in SLOS. Each of 4096 entries are set in lines 3 ~ 7. If current entry of first translation table is not for heap memory region, it sets entry value with the associated page table’s base address along with property bits. If current entry is for heap memory region, it sets the entry value with simply 0 for generating *section translation fault* and demands pages later. The implementations for this are in init_pgt() in *kernel/core/mm.c*.

5.5.3 Initialization of Second Level Translation Table

Bit[31:12] in the second level translation table points to the start address of 4KB page frames in the memory which is the base address of final location. The lower 12 bits (bit[11:0]) sets the properties of the physical frames.

The descriptor setup in second level translation table is the latter part of init_pgt() function. As described in the memory map, the first 3GB (0x0000_0000 ~ 0xBFFF_FFFF) region is directly mapped address. That virtual address of that region is exactly same as the physical address. The 1GB DDR RAM address, GP master AXI interface 0/1 address fall into this region. The property set value for this region is 0x472 which means:

1) *Bit[0], XN (eXecution Never)*
   This bit must be 1’b0.

2) *Bit[1], Descriptor type*

1’b0 for large page and 1’b1 for small page. If a descriptor is for a Large Page table, the bit[1:0] must be 2’b01. If a descriptor is for a Small Page table, the bit[1] is 1, and bit[0] is for *XN.*

3) *Bit[2], B (Bufferable)*
This bit is 1’b0. Not bufferable. Normally, this bit and bit[3], *Cacheable* bit, could be set according to the memory region types we covered in chapter 5.3.3. Recall that there are 3 memory region types which are *Strongly-ordered memory, Device memory, and Normal memory*. Some of the 1GB RAM memory region (0x0000_0000 ~ 0x3FFF_FFFF) could be set as normal memory region which allows the out-of-order access. The I/O device memory that is used by *memory mapped IO* could be bufferable but not cacheable. Some CPU private registers are *Strongly-ordered* memory region that doesn’t allow any out-of-order sequence. For simplicity, SLOS sets all memory region as non-cacheable and non-bufferable. So, the value for this bit[2], *Bufferable,* is 0.

4) *Bit[3], C (Cacheable)*
As explained above, this bit is set as 0, non-cacheable.

5) Bit[5:4], AP[1:0] (Access Permission)
This bit must have 2’b11 for read/write full access. SLOS doesn’t support the *access flag and access flag fault.* This is set in the SCTLR.AFE bit.

6) Bit[8:6], TEX[2:0] (Type EXtension Field)
This bit is used together with *C* and *B* bits to set the memory region attributes. This bit is set as 3’b001 with *0* for *C* and *B* bits for *Outer and Inner Non-cacheable* for normal memory region.

7) Bit[9], AP[2] (Access Permission)
This bit is 1’b0 for R/W full access.

8) Bit[10], S (Shareable)
This bit determines whether the addressed region is shareable memory. Let’s just set it as 1’b1.

9) Bit[11], NG (Non-Global)
Let’s set it as 1’b0 for setting the translation global, meaning the region is available for all processes and don’t care the ASID.

These values could be changed to different usage. For example, the memory regions for CPU private memory region must be marked as Device or Strongly-order memory type [3]. In this case, the bit[8:6], *TEX* bit, must be set as 3’b000. The property value becomes 0x432 for the CPU private memory range 0xF890_0000 ~ 0xF8F0_2FFF.

Now the second level table entries for the first 3GB region can be as below pseudo code.

```
1    ppage_tbl = (unsigned int *)KERN_PGT_START_BASE;
2
3    for i = 0 ... (3 * 1024 * 256) do
4            ppage_tbl[i] = (i * 4096) | 0x472
```

Line 1 sets the second level translation table start address. The start address of second translation table is defined as 0x0021_A000. Line 3 ~4 sets the 3GB memory region 0x0000_0000 ~ 0xBFFF_FFFF as a directly mapped address with normal memory type. See the property bits, bit[11:0], has the 0x472.

The 1GB memory region of kernel’s virtual address region (0xC000_0000 ~ 0xDFFF_FFFF) is remapped to DDR RAM address 0x0000_0000 ~ 0x1FFF_FFFF. This is because the virtual address starts from 0xC000_0000 but their loaded address is in the RAM address starting from 0x0000_0000. This is easily done by below codes.

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

These two lines sets bit[31:12] in the second level descriptor to start from 0x0000_0000. This rempas the virtual address 0xC000_0000 ~ 0xDFFF_FFFF to 0x0000_0000 ~ 0x1FFF_FFFF.

The next 1GB address (0xE0000000 ~0xFFFFFFFF) is also directly mapped address except the CPU private address. The Cortex-A9 MPCore-TRM document [3] 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] (TEX[2:0]) should be 3’b000. Then, the property bits for CPU private region is 0x432. The CPU private region is defined in the Zynq7000 TRM document as 0xF890_0000 ~ 0xF8F0_2FFF. The memory address and property setup for CPU private register are set by below.

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

These entries for CPU private memory region is alos directly mapped address because those address range is already reserved by design. We can’t change the address. And the property must set properly. This region is set as Strongly-ordered region in SLOS, which means it doesn’t allow an out-of-order access, and it is non-cacheable and non-bufferable memory region.

5.5.4 MMU Programming

After creating the address translation tables in init_pgt(), the flow returns back to init_mm.S. We need to finish the rest part of MMU configuration properly in figure 5-20. The CP15 registers are the places for setup the MMU and the caches. Chapter B3.17.2 in ARM TRM document summarizes the description of CP15 registers.

First, we just created the translation tables in previous chapter, and now, we need to invalidate the TLBs to refresh with new table entries. Refer the ARM TRM document chapter B3.18.7 to invalidate the TLB with the ARM ISA instruction.

```
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
```

The upper lines of code invalidate instruction TLB and data TLB, but ARM deprecates the use of them and recommends that software always uses the unified TLB operations [2]. So only line 8 ~9 is valid for invalidate the TLB.

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

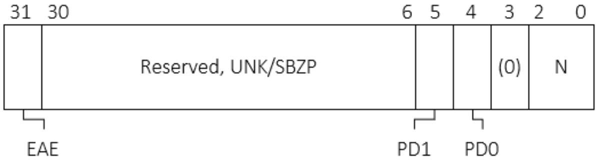


Figure 5-21: TTBCR register bit description

1) 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.

2) Bit[30:6], Reserved
   Should Be Zero.

3) Bit[5], *PD1*
   0: perform translation table walk using TTBR1.
   1: for generating a Translation Fault when a TLB miss on an address translated using TTBR1. No translation table walk is performed.

4) Bit[4], *PD0*
   Enable/Disable the translation table walk using TTBR0. The meaning of possible values of this bit are equivalent to those for the PD1 bit.

5) Bit[2:0], *N*
   This bit Indicates the width of base address held in TTBR0. This value N also determines:
   a) Whether TTBR0 or TTBR1 is used as the base address for translation table walk.
   b) the size of translation table pointed by TTBR0.
   N can take any value from 0 to 7.

Let’s look into the least significant N bits in more detail. This description comes from reference [5]. An additional potential difficulty associated with managing multiple applications with their individual translation tables is that there could be multiple copies of the L1 translation table, one for each application. Each of these will be 16KB in size. Most of the entries will be identical in each of the tables, as typically only one region of memory will be task-specific, with the kernel space being unchanged in each case. Furthermore, if a global translation table entry is to be modified, the change will be required in each of the tables. To help reduce the effect of these problems, a second translation table base register is provided.

CP15 contains two Translation Table Base Registers, TTBR0 and TTBR1. A control register (the TTB Control Register) is used to program a value in the range 0 to 7. This value (denoted by N) tells the MMU how many of the upper bits of the virtual address it must check to determine which of the two TTB registers to use. When N is 0 (the default), all virtual addresses are mapped using TTBR0. With N in the range 1-7, the hardware looks at the most significant bits of the virtual address. If the N most significant bits are all zero, TTBR0 is used, otherwise TTBR1 is used. For example, if N is set to 7, any address in the bottom 32MB of memory will use TTBR0 and the rest of memory will use TTBR1. As a result, the application-specific translation table pointed to by TTBR0 will contain only 32 entries (128 bytes). The global mappings are in the table pointed to by TTBR1 and only one table must be maintained. When these features are used, a context switch will typically require the operating system to change the TTBR0 and ASID values, using CP15 instructions. However, as these are two separate, non-atomic operations, some care is required to avoid problems associated with speculative accesses occurring using the new value of one register together with the older value of the other. OS programmers making use of these features should become familiar with the sequences recommended for this purpose in the ARM Architecture Reference Manual.

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. TTBR0 and TTBR1 holds the base address of translation table 0 and 1. If N == 0, then only TTBR0 is used for translation table walk. 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 the value N increases, the range that TTBR1 covers is increasing like below figure 5-22.

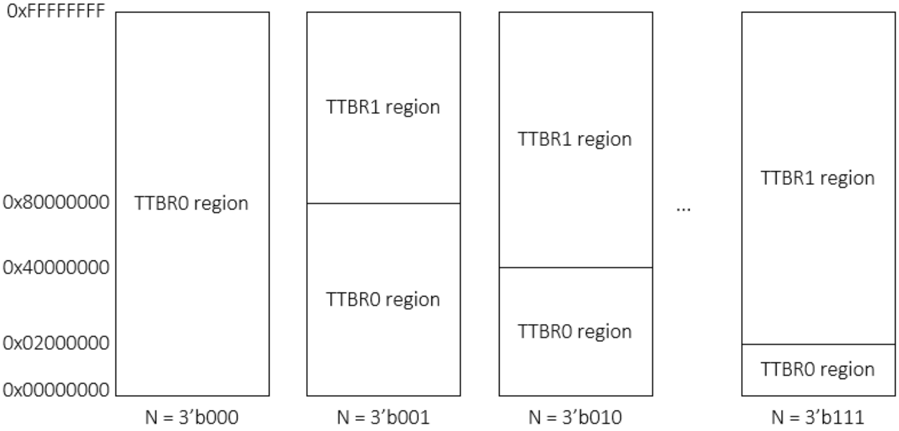


Figure 5-22: Address range covered by TTBR0 and TTBR1 as N increases

In figure 5-22, the total 4GB address space can be split into the TTBR0 and TTBR1. Since TTBR0 covers the lower address space, TTBR0 can be used for user address space. Since

TTBR1 covers the upper address space, TTBR1 can be user for the kernel address space. Possible combinations of TTBR0 and TTBR1 are depicted in figure 5-22. But the kernel virtual address in SLOS starts from 0xC000_0000 and this scheme (using TTBR0 for user, TTBR1 for kernel) can't be accomplished in SLOS. SLOS needs to split its 4GB address space as 3GB for user and 1GB for kernel which is 3:1 ratio. Then, SLOS uses only TTBR0 and value N is set as 3'b000. So, the final value to set TTBCR is 0x0000_0000.

Next, set the TTBR0 with the address of the first level translation table's base address. That is 0x000_0000 in SLOS. SLOS puts the page directory from 0x0000_0000 as below.

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

Finally enable the MMU with System Control Register (SCTLR) and jump to the kernel reset handler.

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

Line 2 ~ 7 enables data cache, instruction cache and disable access flag. After setting up the translation table, enabling MMU in line 7 makes all the logical addresses translated through the translation tables which are set in chapter 5.5.2 and 5.5.3.

Now, we have finished the setting up of Small Page translation table walk with MMU. Line 9 ~ 10 jumps to the reset_handler() function and this reset handler located at 0xC010_1000 as specified in the linker script. After this, the SLOS kernel takes over the control.

5.6 SLOS Heap Manager

There are two major jobs that SLOS's memory manager mostly works on.

1) Establishes translation table and enables the virtual address space by the translation

table walk in MMU.

2) Allocates / Frees the heap memory by implementing the *Demand Paging*.

We just finished he first part that is about how to implement the virtual memory space, and how to map the virtual address space to physical address space in chapter 5.3, 5.4 and 5.5. Once we set up those mappings and enable translation table walk, there are not many things that operating system needs to do for them: the MMU hardware takes care of them. From this chapter, we will dig into the second part: memory manager on heap memory. It is the part that is accomplished by software - the SLOS kernel.

SLOS memory manager for heap is composed of *frame pool, page pool* and *virtual memory pool*. Those manages on memory frame, page table, and virtual memory pool and their implementations are in frame_pool.c, page_table.c and vm_pool.c file. Frame pool in frame_pool.c splits the RAM memory with frames which is 4KB contiguous chunk of memory. Virtual memory pool in vm_pool.c maintains the metadata for virtual memory pool. page_table.c handles the Demand Paging while it traverses the table maps between virtual memory and memory frame.

As we mentioned, SLOS memory manager works on the heap memory allocation and free. SLOS kernel has 64MB heap for an application usage such as allocating *task_struct* instance, red-black tree node, *timer_struct* instance and so on. Unlike the stack that is set by the stack pointer (r13), heap memory is requested by software and also must be managed by software. The heap memory allocation in SLOS is called a *lazy allocator*. The memory manager doesn't allocate the actual memory frames when a kmalloc() is called but it just updates the meta data of virtual memory pool. When there is a real access to the allocated memory later, then the memory manager will allocate the memory frames in the heap. This is called a lazy allocator or *Demand Paging*.

Memory manager is using frames to manage its physical memory. Frame is a 4KB size memory region and is a minimum size of memory allocated at a time. For example, even though there is a need to use 2KB heap memory, memory manager still allocates one 4KB frame for 2KB heap requirement and the last 2KB memory is lost and not used.

The physical memory address is calculated from frame number as below definition. Parameter *X* is a frame index number and the physical address is calculated as 4KB * frame_number.

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

SLOS has 64MB heap ranged from 0xC400_0000 to 0xC800_0000. This heap memory is managed by memory frames and allocated by calling kmalloc(). When kmalloc() is called, memory manager adds a region descriptor (*struct region_desc)* to the meta data list of virtual memory pool but not yet actually allocate the physical heap memory frames. The meta data of virtual memory pool is placed in a 4KB memory frame and is a linked list of region descriptor instances. In the lazy allocator, memory manager doesn't allocate the physical memory until there is an access to that the memory. When there is an access to the allocated heap occurs, a translation fault happens because the translation table doesn't have the entries for that region. The translation table fault happens when the value of the last two bits (bit[1:0]) is 2'b00 which means an invalid entry. The *data_abort_handler* in exception vector must handle this translation fault. It will allocate a physical frame from frame pool of the heap and update the translation table entry for this newly allocated frame region. This is how a lazy allocator or a Demand Paging works in SLOS. We will handle this in a separate chapter in chapter 5.7. In this chapter, we will discuss the frame pool manger and virtual memory pool manager only.

5.6.1 Memory Frame Pool Manager

Memory frame pool manager assigns one 4KB frame for a frame pool bitmap at the address of 0x0021_5000. One single bit of this bitmap represents the allocation status of the 4KB physical frame in a memory. The total memory size that one bitmap frame can cover is 128MB (4K * 8 * 4KB) which is ranged from address 0x0000_0000 to 0x0800_0000. The first half of the memory region is a pre-allocated region for kernel usage. This lower 64MB region is already reserved for kernel text, data, stack, page tables and so on. This region should not be allocated or freed by the memory manager.

The upper 64MB covers from 0x0400_0000 to 0x0800_0000 is a kernel heap memory region which is allocated or freed by the memory manager. Since the offset between virtual address and physical address is 0xC000_0000 (see the linker script in chapter 5.4.1), the virtual address for this heap region is 0xC400_0000 to 0xC800_0000. 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 heap memory. The kmalloc() function calls the virtual memory pool's allocate() function. The allocate() function is a lazy allocator which doesn't allocate the real physical memory frame until there is an access to the allocated memory. What it does is to create a new virtual memory descriptor and returns the address of it. kfree() gets the start address of that memory as its parameter. The start address is the address value returned from kmalloc() function. kfree() calls the virtual memory manager's release() function which release the virtual memory descriptor and delete the entry in the page table.

Below figure 5-23 shows the memory frame layout of one frame bitmap region (0x0000_0000 ~ 0x0800_0000). Each square block is a 4KB size frame. The layout of the lower

64MB frames (0x0000_0000 ~ 0x0400_0000) is done from many different places such as linker script, reset handler, memory manger, forkyi(), and so on. If you are not clear on this, refer the figure 4-2. These frames are predefined and are not allocated or freed.

The last 64MB region is used for heap memory region. This heap memory region is not yet allocated and the translation table doesn't have any entries for this region at the begining. Only the kernel's frame pool manager splits these regions with frames.

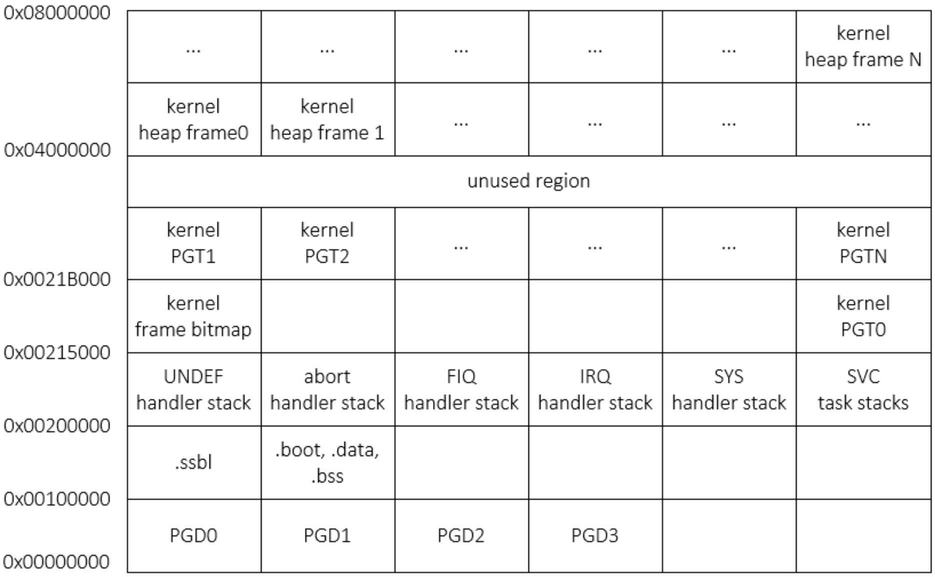


Figure 5-23: Memory frame layout in 0x0000_0000 ~ 0x0800_0000

This 128MB memory frames are mapped into two places in virtual memory address (see the figure 5-19). First, it is directly mapped to the virtual address from 0x0000_0000 to 0x0800_0000. In this case, the virtual address is same as its physical address. Actually, the first 1GB virtual address is directly mapped to its physical address. Second, the virtual address from 0xC000_0000 to 0xC800_0000 is also mapped to this 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'* command, you can see the symbol location is starting from 0xC010_1000 which is set in the linker script.

5.6.2 Virtual Memory Pool Manager

Virtual memory pool is the pool that holds the meta data of the memory allocated from kmalloc(). kmalloc() calls only the allocate() function of virtual memory pool and the allocate() assigns one region descriptor to the virtual memory pool. For this, the first heap memory frame is dedicated to store this meta data. The structure of this meta data looks like below.

```
/* region descriptor structure */
struct region_desc {
      unsigned int startAddr;
      unsigned int size;
      struct region_desc *next;
      struct region_desc *prev;
};
```

This region descriptor has *startAddr* member for the base address of allocated heap memory, the *size* member variable is for the allocated memory size. The region descriptor instances are doubly linked list with the next region descriptor. Figure 5-24 is an example of the description of the meta data frame storing the region descriptor instances and its relation to the heap allocation.

The allocate() function finds an empty space for the amout of *size* parameter and adds another region descriptor for that region to the end of the linked list and returns. The virtual memory manager just updated its meta data frame and returns. There are no actual memory allocations at this moment. The real heap memory frame allocation will be done when there is a memory access. This lazy allocator in virtual memory manager will be covered in next chapter. In this chapter, let's remember that the virtual memory allocator just updates its meta data frame and returns - no real memory allocation.

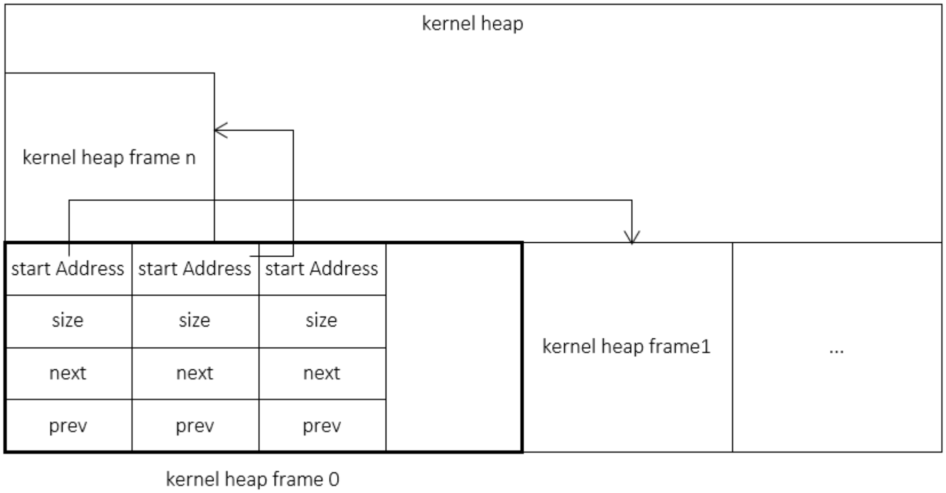


Figure 5-24: Virtual memory descriptor in virtual memory manager

Since only kernel heap frame 0 is assigned to the container for this region descriptor in

SLOS, there is a limit on the number of this linked list. SLOS memory manager even doesn't check the limit number of this region descriptor. We are good for now because SLOS doesn't a commercial operating system. We can add the checking routine for the limitation on the region descriptor number. But now, it's OK. The maximum number of region descriptor for the kernel heap can be calculated as below.

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

The size of *struct region_desc* must be 16 bytes and the size of one frame is 4KB. Then, the maximum number of calling kmalloc() without calling kfree() is limited by 256 times. Even there are still free heap memory space, calling kmalloc() more than this number could break the SLOS. Nonetheless, this is still enough for testing the SLOS's virtual memory manager. Since kfree() removes the region descriptor entry in the meta data frame, this limit number is the net number of calling kmalloc() without kfree(). Later, we will run a test that infinite number of kmalloc() and kfree() in one of CFS task.

5.7 Demand Paging Implementation

As we discussed before, the virtual memory pool is a lazy allocator that the real memory allocation happens when there is an access to the heap memory. Following figure 5-25 describes the concept of the lazy allocator in SLOS.

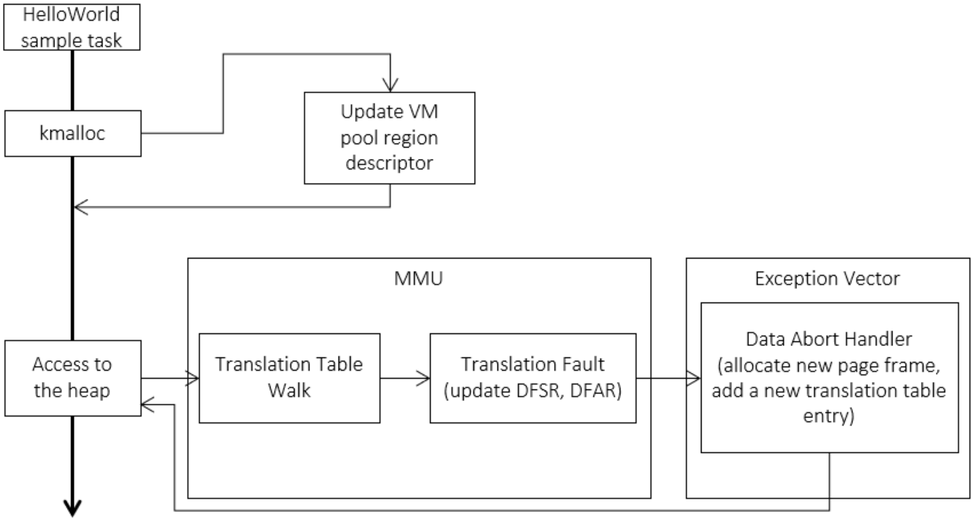

HelloWorld sample task
kmalloc
Update VM pool region descriptor
MMU
Exception Vector
Access to the heap
Translation Table Walk
Translation Fault (update DFSR, DFAR)
Data Abort Handler (allocate new page frame, add a new translation table entry)

Figure 5-25: Lazy allocator (Demand Paging) flow in SLOS

When some application or somewhere in SLOS kernel calls a kmalloc() to use a heap memory, the virtual memory manager updates its region descriptor meta data and returns the start address for that region descriptor. The application flows down the road until there is a real access to that heap memory. When the access happens, the MMU generates a translation fault because there is not an entry in the translation table for that memory region. There is another need in exception handler to handle this data abort. The data abort handler adds a new translation table entry and allocate a physical memory frame for this into the frame bitmap. We will implement the data abort handler for the translation fault in chapter 5.7.2.

5.7.1 Fault Types in ARM

Although we already covered the different types of faults in chapter 5.3.7, it is worth recalling the fault types because the demand paging depends on one of those fault types. Below is the summary of ARMv7 fault types.

1) *Alignment Fault*
   The ARMv7 memory architecture requires support for strict alignment checking. This checking is controlled by SCTLR.A bit. In addition, some instructions do not support unaligned accesses, regardless of the value of SCTLR.A. An alignment fault can occur on an access for which the MMU is disabled.

2) *Translation Fault*
   There are two types of translation fault: Section and Page.
   a) *Section translation fault* is generated if the level one descriptor is marked as invalid. This happens if bit[1:0] of the descriptor are 2'b00.
   b) *Page translation fault* is generated if the level two descriptor is marked as invalid. This happens if bits[1:0] of the entry are 2'b00.

3) *Access Flag Fault*
   If SCTLR.AFE bit is set, the access flag fault is enabled. If access flag fault is enabled, the AP[0] in the translation descriptor is used for access flag. Access flag bit is used to designate whether there is a memory access to the associated address space or not. When SCTLR.AFE is enabled and access flag is 0, then access flag fault is generated.

4) *Domain Fault*
   When using the Short-descriptor translation table format, a domain fault can be generated at the first level or second level of lookup. The reported fault code identifies the lookup level. The conditions for generating a domain fault are:
   a) First level domain fault

When a first-level descriptor fetch returns a valid Section first-level descriptor, the domain field of that descriptor is checked against the DACR (Domain Access Control Register) register. A first-level Domain fault is generated if this check fails.

b) *Second level domain fault*
When a second-level descriptor fetch returns a valid second-level descriptor, the domain field of the first-level descriptor that required the second-level fetch is checked against the DACR, and a second-level domain fault is generated if this check fails.

5) *Permission Fault*
Permission fault is generated if the checking of access permission bits fails. Access permission bits are AP[2:0] when access flag feature is not enabled and AP[2:1] when access flag is enabled. AP[0] bit is used for access flag in the latter case.

SLOS doesn't use anything but the translation fault. SLOS disables SCTLR.AFE to disable access flag fault, AP[2:0] is set as 3'b011 for full read/write access, set domains as manager for the 0xF domain which is used for SLOS domain setting. Refer the figure 5-8 for fault checking sequence.

5.7.2 Data Abort Handler

We have to add another exception handler to handle the fault during the translation table walk. This translatioin fault is handled in the *data_abort_handler* of the exception vectors. The data_abort_handler has an offset of 0x10 from the base of exception vector. Now, the data_abort_handler in the kernel is changed as below.

```
data_abort_handler:
    sub     r12, r14, #8
    msr     cpsr_c, #MODE_SVC | I_BIT | F_BIT
    stmfd   sp!, {r0-r11, r12}
    push    {r14}
    mrc     p15, 0, r0, c5, c0, 0
    bl      platform_data_abort_handler
    pop     {r14}
    mrs     r0, CPSR
    bic     r1, r0, #I_BIT|F_BIT
    msr     cpsr_c, r1
    ldmfd   sp!, {r0-r11, pc}
```

In line 2, it calculates the return address by subtracting the current *lr* register (r14) by 8. Line 3 changes the processor mode to SVC mode and disables the IRQ and FIQ interrupts by setting the *I-bit* and *F-bit* in the CPSR register. Line 4 ~5 saves the current task's context to its stack. This is the task that gets the data abort exception. The line 6 reads the *DFSR (Data Fault Status Register)* from CP15 to register 0 (r0) and deliver this value to the function *platform_data_abort_handler* as the first parameter. Refer the ARMv7 reference manual B3.17.1 for the assembler code of this. The DFSR register holds the last data abort status information. Below figure 5-26 shows the DFSR register's bit information.

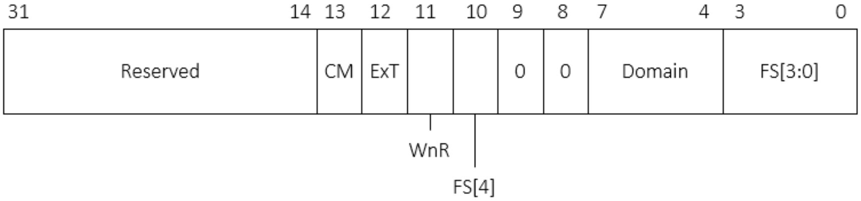


Figure 5-26: DFSR register bit information

1) *Bit[31:14]: Reserved*

2) *Bit[13], CM (Cache Maintenance)*
If implementation includes the LPAE, this bit means cache maintenance fault.
0: for an abort not caused by a cache maintenance operation.
1: for an abort caused by a cache maintenance operation.
If implementation doesn't include LPAE, this bit is reserved.

3) *Bit[12], ExT (External abort Type)*
This bit can provide an IMPLEMENTATION DEFINED classification of external aborts. In am implementation that doesn't provide any classification of external aborts, this bit is UNK/SBZP.

4) *Bit[11], WnR (Write not Read)*
On a synchronous exception, indicates whether the abort was caused by a write instruction or by a read instruction. The possible values of this bit are:
0: Abort caused by a read instruction.
1: Abort caused by a write instruction.
For synchronous faults on CP15 cache maintenance operations, including the address translation operations, this bit always returns a value of 1. This bit is

UNKNOWN on:
An asynchronous Data Abort exception, or a Data Abort exception caused by a debug exception.

5) *Bit[10], Bit[3:0], FS (Fault Status)*
This bit shows the different types of fault status. For the valid encodings of these bits when using the Short-descriptor translation table format, see the table 5-4. All encodings not shown in the table are reserved.

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	

Table 5-4: Fault Status bit information

6) *Bit[9], LPAE (Reserved UNK/SBZP)*
On taking a Data Abort exception, this bit is set to 0 to indicate use of the Short-descriptor translation table formats. Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation.

7) *Bit[8]: UNK/SBZP*
Reserved.

8) *Bit[7:4], Domain*
   The domain of fault address. ARM deprecates any use of this field. This field is UNKNOWN on a below data abort exception:
   Debug exception or Permission fault in an implementation includes the LPAE.

SLOS uses only a *Fault Status* bit. All other bits are not used in the SLOS's data_abort_handler. The fault status bits are used for SLOS to figure out the translation fault type. Table 5-4 lists up all fault status bit encodings. In this many fault status type in the list, SLOS supports only the translation fault occurring in the MMU. So, the bold box in the table is the only faults that SLOS implements. The first level translation fault is called a *section translation fault* and the second level translation fault is called *page translation fault*. Line 6 reads the DFSR register and puts it into the register for the first parameter of callee function (r0), the line 7 jumps to the platform_data_abort_handler() in *kernel/exception/faults.c.*

The platform_data_abort_handler() checks its first argument which has the DFSR register's value and calls a proper abort handler. The platform_data_abort_handler() looks like below.

```
if ((dfsr & TRANSLATION_FLT_PG) == TRANSLATION_FLT_PG) {
        handle_fault();
} else if ((dfsr & TRANSLATION_FLT_SEC) == TRANSLATION_FLT_SEC) {
        handle_fault();
} else {
        abort();
}
```

This routine is checking the first argument value with a corresponding fault status mask. The mask values are as below and coming from the table 5-4.

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

As you can see, there is only two checking routines for section translation fault and page translation fault. If the fault satus is coming from either section translation fault or page translation fault, it calls the handl_fault() function. The handle_fault() function will be explained in the next chapter. If the data abort comes from other places, SLOS just spinning forever in the abort() function as below. SLOS doesn't know how to handle these types of faults.

```
void abort()
{
        xil_printf("data abort exception!!\n");
        for(;;);
}
```

5.7.3 Page Fault Handling and kfree()

When the platform_data_abort_handler() checks the current fault type is one of the translation faults, section or page, in MMU, it will call a handle_fault() function in page_table.c. Below is the code of handle_fault() function.

```
1     void handle_fault(void)
2     {
3         asm ("mrc p15, 0, %0, c6, c0, 0" : "=r" (pfa) ::);
4         asm ("mrc p15, 0, %0, c2, c0, 0" : "=r" (pda) ::);
5
6         pgdIdx = ((unsigned int)pfa & 0xFFF00000) >> 20;
7         pde = (unsigned int *)(((unsigned int)pda) + (pgdIdx << 2));
8
9         if ((*pde & 0x00000003) == 0x0) {
10            *pde = (KERN_PGT_START_BASE + ((pgdIdx << 8 ) << 2)) | 0x1E1;
11        }
12
13        frameno = get_frame(pcurrentpgt->pframepool);
14        frame_addr = (unsigned int *)FRAMETOPHYADDR(frameno);
15
16        pgtIdx = ((unsigned int)pfa & 0x000FF000) >> 12;
17        pte = (unsigned int *)(KERN_PGT_START_BASE + ((pgdIdx << 8) << 2) + (pgtIdx << 2));
18        *pte = ((unsigned int)frame_addr | 0x472);
19    }
```

What handle_fault() function does is to insert a new entry for the fault address into the translation tables. First, handle_fault() reads the fault address in the line 3 into a variable *pfa*. It also reads a translation table base address from TTBR0 register in the line 4 and save it into variable *pda*. Refer the ARMv7 reference manual B3.17.1 for the assembler code of this. This assembler code is embedded into the normal C programming language by using an *asm* keyword. We call it an *inline assembler*. Refer document [4] or you can find inline assembler descriptions simply by searching the web.

Now that we know the fault address, we can calculate the entry of it in the first translation table. This is done in line 6 and 7. Line 9 checks the last 2 bits. The description about these 2 bits in the first level descriptor is described in chapter 5.3.8.1. If these two bits are 2'b00, it means there is no entry in the first translation table: fault type is a section translation fault. The fault type also can be checked by using the fault type bits in the DFSR register. If there is no entry, line 10 creates a new section entry for the fault address. Line 13~14 gets a new memory frame from frame pool. The get_frame() function finds out a free frame in the frame bitmap and returns it. It also sets one bit in the frame bitmap for this. Line 16 and 17 calculates the page entry (second level translation descriptor) about the fault address. Then line 18 assigns the base address of allocated new memory frame to this page entry with property bits.

Notice that the base address of memory frame allocated by get_frame() in line 13 can be placed at any address in the memory. In other words, the physical memory address is independent of the virtual address. For example, let's assume process A's the virtual address 0xC400_1000 points the physical location in 0x0400_1000.Then, other process B which has its own translation tables can have the same virtual address (0xC400_1000) but it points to a different physical location like 0x0500_1000. By using this, each process can have separate address map, has its own 4G address space working independently and can't access other process' memory space if it is not shared. This memory security among different processes is another advantage of using MMU and virtual address we talked before.

So far, we discussed how the kmalloc() works with demaing page feature depicted in figure 5-25. kfree() is an opposite function of kmalloc(). kfree() deletes the region descriptor in the virtual memory pool and calls below free_page() function to free the real frame, update bitmap frame and delete the translation table entry.

```
1    void free_page(unsigned int freedAddr)
2    {
3            asm ("mrc p15, 0, %0, c2, c0, 0" : "=r" (pda) ::);
4
5            pgdIdx = ((unsigned int)freedAddr & 0xFFF00000) >> 20;
6            pgtIdx = ((unsigned int)freedAddr & 0x000FF000) >> 12;
             pte = (unsigned int *)(KERN_PGT_START_BASE + ((pgdIdx << 8) << 2) +
7            (pgtIdx << 2));
8
9            frame_addr = (unsigned int *)(*pte);
10          frame_num = (unsigned int)(frame_addr) >> 12;
11
12          if (frame_num >= HEAP_FRAME_START &&
13            frame_num < HEAP_FRAME_START + HEAP_FRAME_NUM) {
14                      release_frame(pcurrentpgt->pframepool, frame_num);
15          } else {
16          }
17
18          *pte = 0x0;
19          asm ("mcr p15, 0, %0, c8, c7, 0" : :"r" (r0) :);
20  }
```

free_page() function starts by reading the TTBR0 register value. It gets the entry address of second level descriptor in line 5 ~ 7 and update the second level descriptor content as 0x0 in line 18. Then, it deletes the physical frame allocation in line 9 ~ 16. The release_frame() in line 14 updates the bitmap by clearing the corresponding bits. The line 19 invalidates the TLB for refreshing the updated entries.

5.7.4 Summary on Demand Paging

Below figure 5-27 is the summary for demand paging implementation in data abort exception vector. When program control flow tries to access the empty entries, the data abort handler figures out how to handle it by reading the DFSR and DFAR registers. Then, if the data abort is caused by demanding new pages, the handle_fault() allocates a new memory frames, update translation table entries and returns the base address of them. After finishing all of these, the data abort exception vector returns the faulted address and proceed the execution again. Now that there are newly generated entries for the faulted address, the program control flow doesn't see the section or page fault again.

SLOS applies the Demand Paging only to its heap memory region. The translation table entries for all other memory regions are set up in the init_pgt() of secondary bootloader. Demand Paging feature in commercial operating system is also used for different purpose. The operating system copies a page from disk to main memory only when there is a page fault (only when there an attempt to access it and that page is not in the memory). This can be used below two scenarios.

1) When memory free space gets short, the operating system swaps the least recently used pages to the disk to free up more space in memory. After a page fault at the moment the swapped page is accessed, the swapped page is loaded to the memory again.

2) When operating system loads a program, it copies pages that the program demands to the memory as opposed to load all pages immediately. There are a couple of advantages with this. As there are more space in memory, more process can be loaded, reducing context switching time. It also less loading latency when a program starts up as less pages are accessed from the disk.

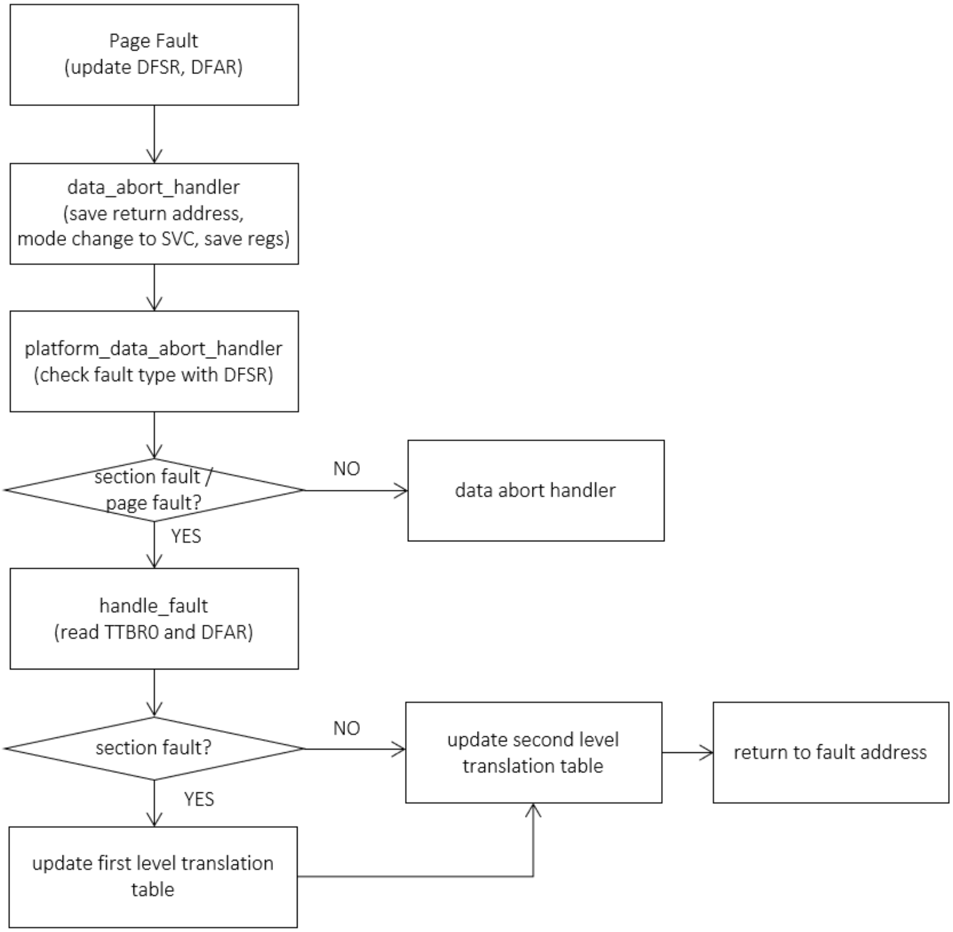

Page Fault
(update DFSR, DFAR)
data_abort_handler
(save return address,
mode change to SVC, save regs)
platform_data_abort_handler
(check fault type with DFSR)
section fault / page fault?
NO
data abort handler
YES
handle_fault
(read TTBR0 and DFAR)
section fault?
NO
update second level translation table
return to fault address
YES
update first level translation table

Figure 5-27: Demand Paging flow in data abort exception

The commerical operating system uses the Demand Paging feature between main memory and disk, but SLOS uses this feature only in heap memory allocation. Nonetheless, the basic concept that uses a page translation fault is still valid.

5.8 Putting it altogether

Up until now, we have successfully demonstrated how to set up the translation tables and how to use page fault to implement the *Demand Paging* feature. After the memory management is implemented, the SLOS has the MMU enabled and runs every program in the virtual address space. The heap allocation in SLOS implements the Demand Paging or Lazy Allocation feature. For example, a kmlloc() is now using Demand Paging for allocating a struct task_struct of a new task in a forkyi() function.

It’s time to verify the lazy heap allocation feature in one of the CFS tasks. The cfs_worker1 task is changed to allocate a 4KB memory (one memory frame) in the heap and accesses the memory for testing the Demand Paging. The body of cfs_worker1 task is changed as below.

```
#define  TEST_KMALLOC_SZ  4096
uint32_t cfs_worker1(void)
{
        uint8_t *pc;
        uint8_t t;
        int i;

        while (1) {
            if (show_stat) {
                    xil_printf("cfs_worker1 is running....\n");
            }

            pc = (uint8_t *)kmalloc(sizeof(uint8_t) * TEST_KMALLOC_SZ);
            if (pc != NULL) {
                    for (i = 0; i < TEST_KMALLOC_SZ; i++) {
                            t = (uint8_t)(i % 256);
                            pc[i] = t;
                    }
            }
            kfree((uint32_t)pc);
            pc = NULL;
        }

        return 0;
}
```

In line 13, cfs_worker1 task allocate a 4KB heap memory in the kernel's virtual memory pool. Since the virtual memory pool is a lazy allocator, this just adds a region descriptor to the meta data of the virtual memory pool and returns the start address of virtual memory pool. kmalloc() still doesn't allocate a physical memory yet. Line 14 ~ 19 tries to write an unsigned char number to each byte of the allocated memory. The first execution of line 16 makes an access to the address that is allocated in line 13. This makes a section translation fault because there is no entry for the first level translation table. This fault will follow the steps in figure 5-27. It first allocates a new entry in the first translation table and in the second translation table and finally the page pool manager allocates a 4KB memory frame in the physical heap memory. Since the first level entry covers 1MB memory region, if cfs_worker1() allocates 4KB heap memory, the access to the next page address will generates only a page fault.

Run *git pull* command to get the latest source code and check out the virtual memory management by running *git checkout SLOS_CH5*. *SLOS_CH5* is a branch name for virtual memory management implementation. Then you can build the source and create the BOOT.BIN to boot up the SLOS in the target board.

Then attach the XSDB debugger and GDB debugger as described in chapter 3.8. After attaching both debuggers, run the *rst* command in the XSDB command terminal. Now the processor is reset and breaks at 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 address of kernel.elf and also the start address of seconary bootloader for the initialization of MMU and page translation tables. Run a *bps 0x100000 hw* command in XSDB terminal and run *c* command in GDB terminal to have the processor continue until the break point at 0x0010_0000. When the processor is stopped at 0x0010_0000, you can start your debugging of secondary bootloader 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 step each assembler code. Then add another break point at init_pgt() function in GDB debugger and continue to browse the source code about translation table generation part. After return from init_pgt() the secondary bootloader code will configure and enable the MMU. If the MMU is enabled, the address space used in the kernel is changed into the virtual address space. You can still step over each line of these and check the processor for debugging.

Let's add another break point at 0xC010_1020 which is the start address of reset_handler of SLOS kernel. This break point should be added in XSDB debugger by using *bps 0xC0101020 hw*. You should know that this address is a virtual address. If you make the processor continue from the secondary bootloader, the processor should break at the reset_handler. You can browse every single line of reset vector handler implementation. Now you can go back to the GDB and add a break point at start_kernel() with a *b start_kernel* command. After the processor breaks at start_kernel, you can go through the start_kernel() code which has all basic initialization functions including process management.

To see the demanding page, add a break point at 0xC010_1010 which is the address of data abort exception handler. You should add this hardware break point from XSDB debugger. If you continue, the break point at data abort handler will be hit after a while because the kmalloc() used in forkyi() will make a first translation fault. The first kamlloc() should generate a section translation fault but from the second kmalloc() 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 Demand Paging implementation. If you add cfs_work1() task from the shell command prompt, this will also generate a page fault and add new pages when demanded. The cfs_worker1 task infinitely calls kmalloc() and kfree() to test the Demand Paging. Figure 5-28 summarizes the steps how to break the processor and where to add the break points to browse the memory management implementations.

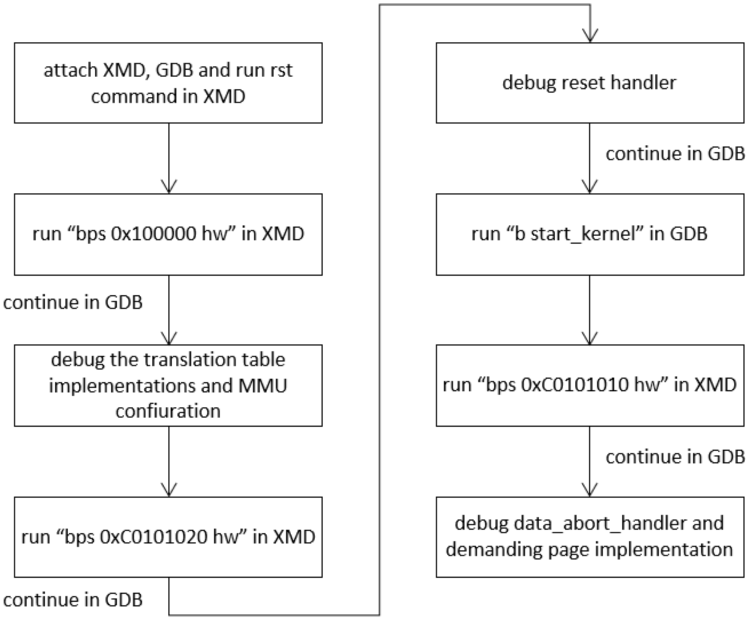

attach XMD, GDB and run rst command in XMD
run “bps 0x100000 hw” in XMD
continue in GDB
debug the translation table implementations and MMU confiuration
run “bps 0xC0101020 hw” in XMD
continue in GDB
debug reset handler
continue in GDB
run “b start_kernel” in GDB
run “bps 0xC0101010 hw” in XMD
continue in GDB
debug data_abort_handler and demanding page implementation

Figure 5-28: Debugging method on translation table generation and demand paging

Memory management means every software including operating system start to use a virtual address space. After enabling MMU, the addresses coming from processor should pass through the MMU’s translation. Using virtual address space needs a special hardware and a complex address interpretation. Then, why modern operating system uses the virtual address? Let’s have a quick discussion about the advantages from using virtual address.

1) Security
   First advantage is a security. Since each program has its own translation tables, there is no way to break into other program's memory if the memory isn't shared. As we discussed, MMU generates a diverse fault based on the page or section properties set in the translation descriptor.

2) Stability
   By separating the address space among different applications, we can protect the kernel's memory space from an instable program's faults. A fault in one application doesn't make serious impacts on the kernel's operation.

3) By using virtual address, each program recognizes all 4GB address space is dedicated to itself. This makes the installed memory capacity larger than its native capacity. If the system runs out of memory, the operating system can swap the memory to disk to free up more space in the memory. So, the secondary storage is used to extend the main memory.

4) Since the physical 4KB pages can be located in any place of physical memory, the virtual address with MMU can reduce the external fragmentation in memory. This is possible because the memory manager doesn't need to allocate a contiguous physical memory. The virtual memory region is contiguous but the physical frames for that virtual memory region can be scattered over the physical memory. As we already implemented, the frame pool allocates new memory frame based on its frame bitmap. These non-contiguous frames are mapped through the translation table.

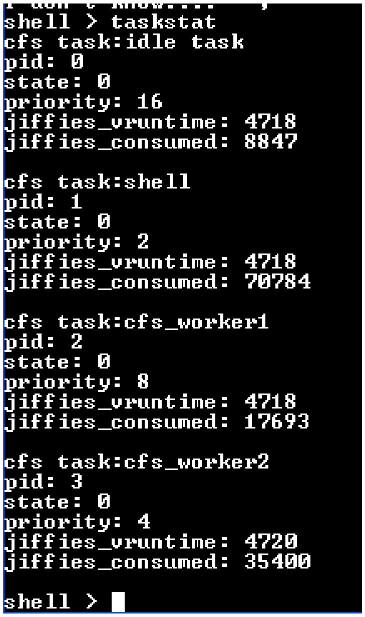
There are also fancy benefits coming from Demand Paging such as fast application loading when it starts up.

The downside of virtual address could be it takes longer to access the physical memory. As described in figure 5-17, there are many steps to get a data from a virtual address. This could be improved by using the TLB.

Next thing is that run a test with cfs_worker1 with repetitive kmalloc() and kfree(). We discussed about the implementation of cfs_worker1. Then, run it to see if the heap allocation and free works fine while switching context. After booting finished, run a *cfs task* command in the shell prompt. Then, wait for a while until there are enough context switching among CFS tasks. After that, run a *taskstat* command to see if CFS Scheduler works fine. Figure 5-29 is a screen capture to see the task statistics while testing kmalloc and kfree. The virtual runtime is pretty fair and the jiffies consumed is still proportional to their priority.

5.9 Summary

In this chapter, we discussed about the address spaces: logical address, virtual address and physical address. The virtual address is activated by enabling MMU. Chapter 5.3 focuses on the ARM MMU. To map the virtual address to physical address, we need to set up the translation tables for the MMU. MMU in ARM has 4 different types of translation formats: Super Section, Section, Large Page and Small Page. We digged into the translation steps in detail especially on Section and Small Page formats.



```
I don't know....
shell > taskstat
cfs task:idle task
pid: 0
state: 0
priority: 16
jiffies_vruntime: 4718
jiffies_consumed: 8847

cfs task:shell
pid: 1
state: 0
priority: 2
jiffies_vruntime: 4718
jiffies_consumed: 70784

cfs task:cfs_worker1
pid: 2
state: 0
priority: 8
jiffies_vruntime: 4718
jiffies_consumed: 17693

cfs task:cfs_worker2
pid: 3
state: 0
priority: 4
jiffies_vruntime: 4720
jiffies_consumed: 35400

shell > █
```

Figure 5-29: CFS task statistics while kmalloc() and kfree() are working

From chapter 5.4, the implementations of SLOS virtual memory management are explained. SLOS memory management supports virtual address, Demand Paging. The virtual address via a Small Page format is implemented in the secondary bootloader. The description on setting up virtual address in SLOS are described in chapter 5.4 and 5.5. SLOS supports the Demand Paging feature only for its heap region. Demand Paging is a lazy allocator for the faulted memory region. SLOS uses page fault for this and adds a new data abort handler in its exception vectors to support Demand Paging. We listed up the benefits of using virtual address through MMU.

In summary, SLOS memory manager supports virtual address space by using a Small Page format and Demand Paging through a page translation fault handler.

References

[1] https://en.wikipedia.org/wiki/Memory-mapped_I/O

[2] ARM, *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*

[3] ARM, Cortex-A9 MPCore Technical Reference Manual

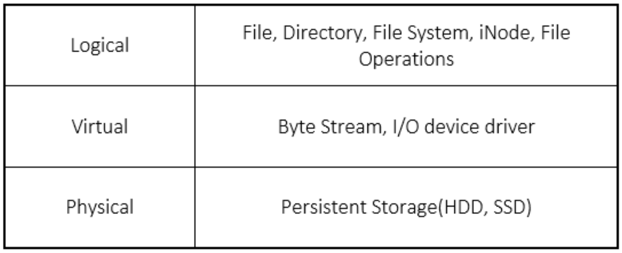
[4] https://en.wikipedia.org/wiki/Inline_assembler

[5] ARM Cortex-A Series Programmer’s Guide

6 Storage Management and Application Loading

6.1 Introduction

When a Zynq7000 chipset is powered on, a simple boot code from the boot ROM first runs. Mostly, all its job is to find a boot device and load custom bootloader into memory. This bootloader loads the operating system image executable file into the predefined memory location and jumps to start address of the operating system. After operating system is loaded from the persistent storage and starts from its start address, its storage manager begins to see the storage as below way.



Logical	File, Directory, File System, iNode, File Operations
Virtual	Byte Stream, I/O device driver
Physical	Persistent Storage(HDD, SSD)

Figure 6-1: Layered model of persistent storage

There are many different types of physical persistent storage such as HDD, SSD, magnetic tape and so on. Data in this storage media are maintained and persistent even after power is down. This is the reason why we store important files such as operating system image, bootloader image into the persistent storage. Persistent storage is normally much bigger but very slower than the main memory. Nowadays, this storage can be used as a secondary memory by swapping pages from the main memory into the storage. We call it a page swap or a demand paging and we covered a little bit on this in chapter 5.

The virtual layer in figure 6-1 is a byte stream stored in the persistent storage or the I/O device driver for the storage. This layer provides virtualized *read/write* functions to access the storage.

The logical layer of storage management includes directory, file, file system and file operations. There are many different types of logical implementations in this layer. For example, there are *FAT, NTFS* in Windows OS, *EXT2, EXT3, EXT4* for Linux and so on. These

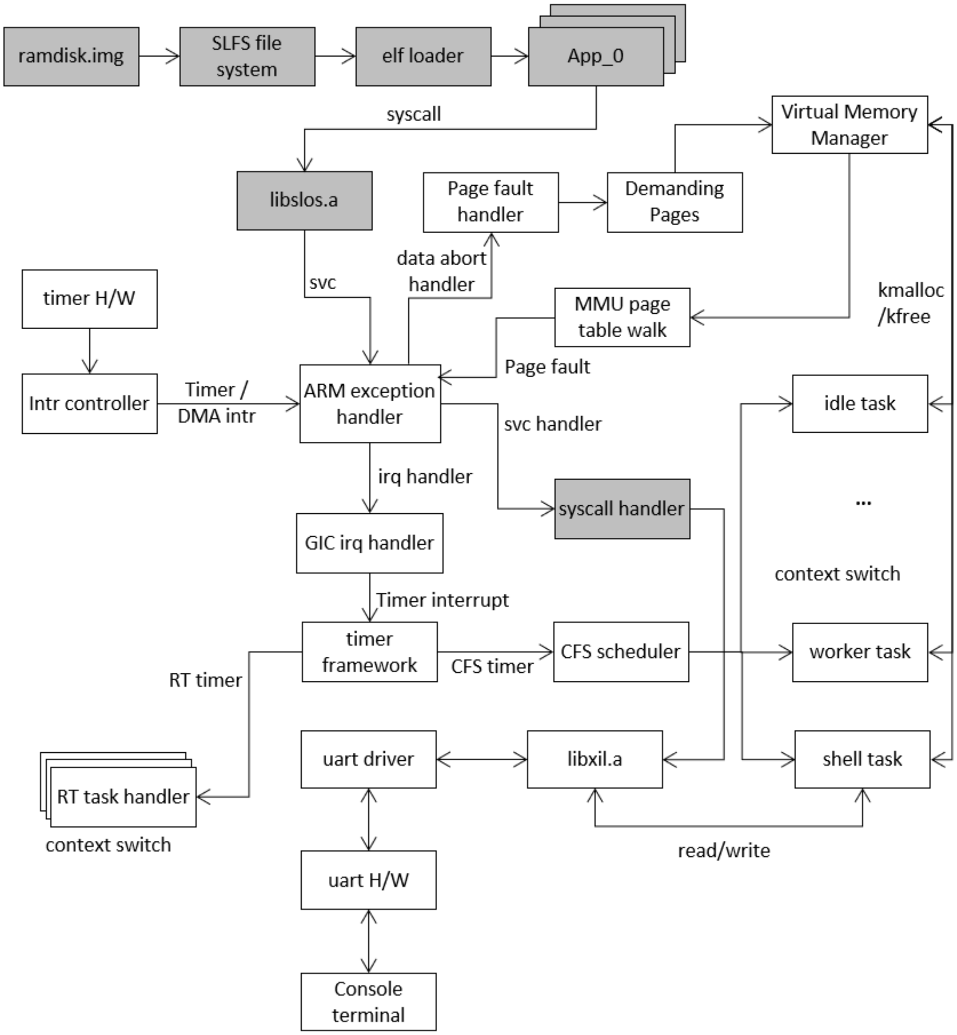


Figure 6-2: SLOS top view when storage management is added

different logical implementations have their own logical views on the physical storage on top of virtual implementations of I/O device driver. These different logical implementations have their own upsides and downsides.

We don't cover each different commercial file systems. That is beyond this book and there are many good articles and papers for this purpose. Instead, we will implement a *Simple and Light File System (SLFS)* to store and load the user-built applications. We will touch a simple but general concept of the file system such as *iNode*. In addition to this simple file system, we will extend the process management in chapter 4 to the user applications by using a *system call*.

After this chpater, the SLOS subsystems look like figure 6-2. The grayed blocks are new blocks added in this chapter. SLOS will have SLFS and *system calls* plus other blocks to coordinate with the user applications. The changes or newly added blocks are:

1) SLFS storage manager is implemented.
2) Build system must be changed to build *libslos.a* library.
3) Build system creates a ramdisk.img to put together all user applications.
4) An *ELF loader (Executable and Linkerable Format)* is implemented.
5) Simple helloworld user applications are added.
6) To handle system calls, a new *syscall_handler()* is added to the SLOS's exception vector.

These blocks are marked within grayed block in figure 6-2. After this chapter, SLOS will have the three basic features (process memangement, memory management, and persistent storage management) of an operating system.

After the SLFS implementation is done, we will use the SLFS to build a ramdisk image containing a simple user application. This application will print a message through a *system call*. Printing a simple 'hello world' message from the user application is not such a simple process. User application should access to system hardware resource and in this case, it is the UART hardware. For demonstration of this simple function, SLOS provides a library for system call, and implements system call handler and an ELF loader. All other blocks in figure 6-2 except SLFS block are related to the system call functions.

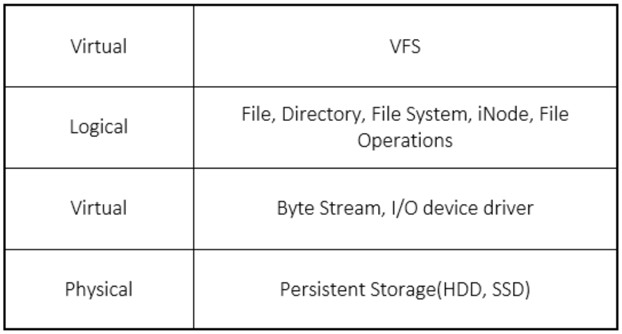
This chapter describes the SLFS implementations in the first half and describes how to use the SLFS in its second half.

6.2 SLFS File System

SLFS is a file system of SLOS. It works on top of ramdisk: the physical layer in figure 6-1. SLFS is using a RAM memory for its storage device which is not actually a persistent storage. But using the RAM memory for storage makes the implementation of virtual layer - I/O device driver- very easy. We can access the disk storage simply by the address which is exactly same as a memory access. In addition, we don't need to consider a blocking I/O access and an interrupt from the storage hardware for the virtual layer implementation.

Ramdisk is widely used in the Linux system. After Linux bootup is finished, the file system in ramdisk is mounted to the *root* directory (/). This *init ramdisk* contains many prebuilt tools and service daemons. SLFS adopts this idea because SLOS doesn't have a complex I/O device driver for the storage and we don't need to waste our time on I/O device driver development and can focus on the logical layer of storage management.

As in figure 6-1, file system is the logical form of storage hardware. But, even one specific operating system can have many types of logical implementations for handling a storage media; for example, EXT2, EXT3, EXT4 in Linux. These different file systems run in the same hardware. The applications running in the highest layer that needs to access these diverse file systems would be in panic if they use all different file system in a different way. For this, each different file system implements the *VFS (Virtual File System)* to abstract the diverse file systems. The Virtual File System (also known as the *Virtual Filesystem Switch*) is the software layer in the kernel that provides the uniform filesystem interface to userspace programs. It also provides an abstraction within the kernel which allows different filesystem implementations to coexist. VFS *system calls (open, stat, read, write, chmod* and so on) are called from a process context [2]. So, in normal, commerical high level operating systems, figure 6-1 has one more layer of virtualization on top of the logical layer implementations as in figure 6-3.



Virtual	VFS
Logical	File, Directory, File System, iNode, File Operations
Virtual	Byte Stream, I/O device driver
Physical	Persistent Storage(HDD, SSD)

Figure 6-3: Storage management layers in commercial OS

SLFS is simple and doesn't have any implementations for VFS. Since SLOS also has only one simple system call, *write,* for printing message to terminal, there is no need to have such VFS system calls for user applications. But this limitation doesn't hurt anything about the SLOS's developement intention.

6.2.1 SLFS Work Flow

SLFS's high level work flow looks like figure 6-4. As in other subsystems, the start_kernel() is the place that SLFS starts from. It contains the routines of initialization, mount and format of the SLFS. After mounted, the SLFS can create new files to read or write to the storage. SLOS supports only ramdisk storage, and the access to the ramdisk is a 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 access to the media storage goes to sleep state and wait until its read/write is done. After read/write is finished, an interrupt fires the task to wake up and run again from the blocked location. In non-blocking access, the read/write to storage returns right after queuing the request and the task keeps running. Since SLFS's read/write is just a memory load/store, which means the task is waiting but not sleeping until the read/write is finished.

SLFS file system stores the meta data of files into the inodes. This meta data is named as *inode* which pointing the data locations in the data block. These inodes are used for the file open/close, read/write access.

Task can create a file, read or write data into the opened file. Currently, the SLFS supports only read and write operations. The real physical read and write happens in its memory read/write device driver. This blocking device driver is just memory access and such simple that we don't need to spend time in developing the storage I/O device driver. Nonetheless, we will implement a simple IO device driver for ramdisk.

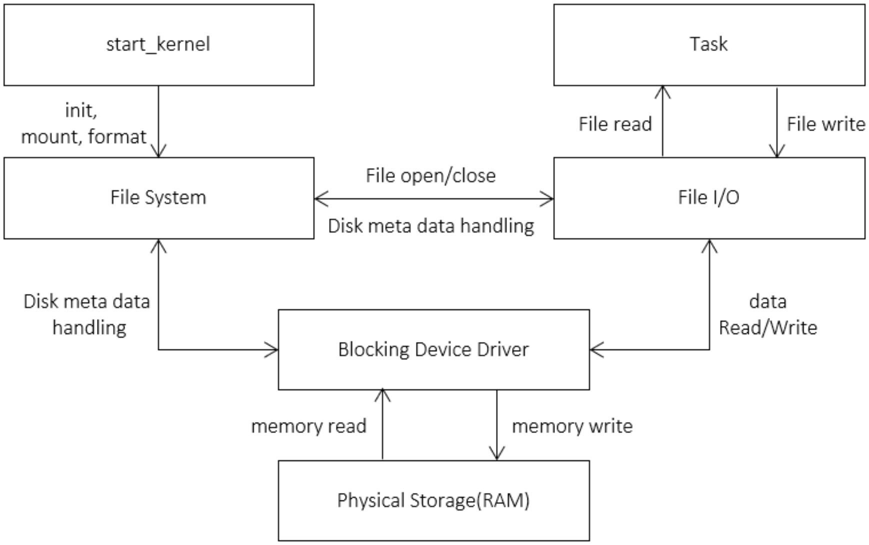


Figure 6-4: High level work flow of SLFS

6.2.2 Source Changes for SLFS

To see the implementations for SLFS, pull the sources from github and checkout the *SLOS_CH6* branch. SLFS itself is not a big change in SLOS but we are going to change the SLOS build sequence to test ramdisk with user applications. Following files are added or modified for SLFS implementation.

1) *kernel/core/file_system.c*

   This file has SLFS's meta data blocks *(super section, inode table, data block bitmap, inode bitmap)* implementations. There are implementations for file system init, mount, format, file creation, data block allocation and so on.

2) *kernel/core/file.c*
This file has SLFS file manipulations such as file open, close, read, write. It uses an inode to represent a file in the file system. Read, write operations use the inode to represent the associated data block locations.

3) *kernel/core/ramdisk_io.c*
This file has ramdisk I/O implementations. This works as an IO device driver. The IO operation is same as a normal memory access.

4) *kernel/inc/file_system.h, file.h,ramdisk_io.h*
These are header files for SLFS functions.

5) *kernel/exception/kernel.S*
The exception vector has a new handler for handling the system call which is offset at 0x8 from exception vector base. This location is for *supervisor call* exception. The new exception handler works for the bridge between user mode application and kernel. We already discussed that user applications can't directly access the system hardware resources. It happens only through the kernel. This system call is a predefined message from user application for the kernel to expose the system hardware resources to user application.

6) *libslos/syscall.S, libslos/print_mesg.c*
These files compose a library for system call. Currently there is only *write* system call in SLOS. This system call can relay the message string from user application to the UART device driver in the SLOS kernel.

7) *apps/helloworld.c*
This is a sample user application which is built into a ramdisk image. This user application is linked with *libslos.a* library to use a *write* system call.

8) *mkfs/mkfs.cpp*
This file is an application tool to build a ramdisk image.

9) *Makefile*
Build process is changed in this chapter. Makefile is changed to build SLOS library, ramdisk image, ramdisk image build tool. Changes in Makefile will be covered in chapter 6.7.1.

6.2.3 Memory Map for Ramdisk

Since ramdisk, as the name says, uses a RAM memory for the storage media, SLOS needs to allocate 4MB memory for the ramdisk. The memory map in figure 5-19 in chapter 5.4.2 is updated as new memory map in figure 6-5. Notice that there is a grayed block for ramdisk storage from 0x0300_0000. This memory region is reserved for ramdisk. Other memory regions remain same as the chapter 5 memory management. Later, we will load user applications from the file in ramdisk image into this region and create SLFS files.

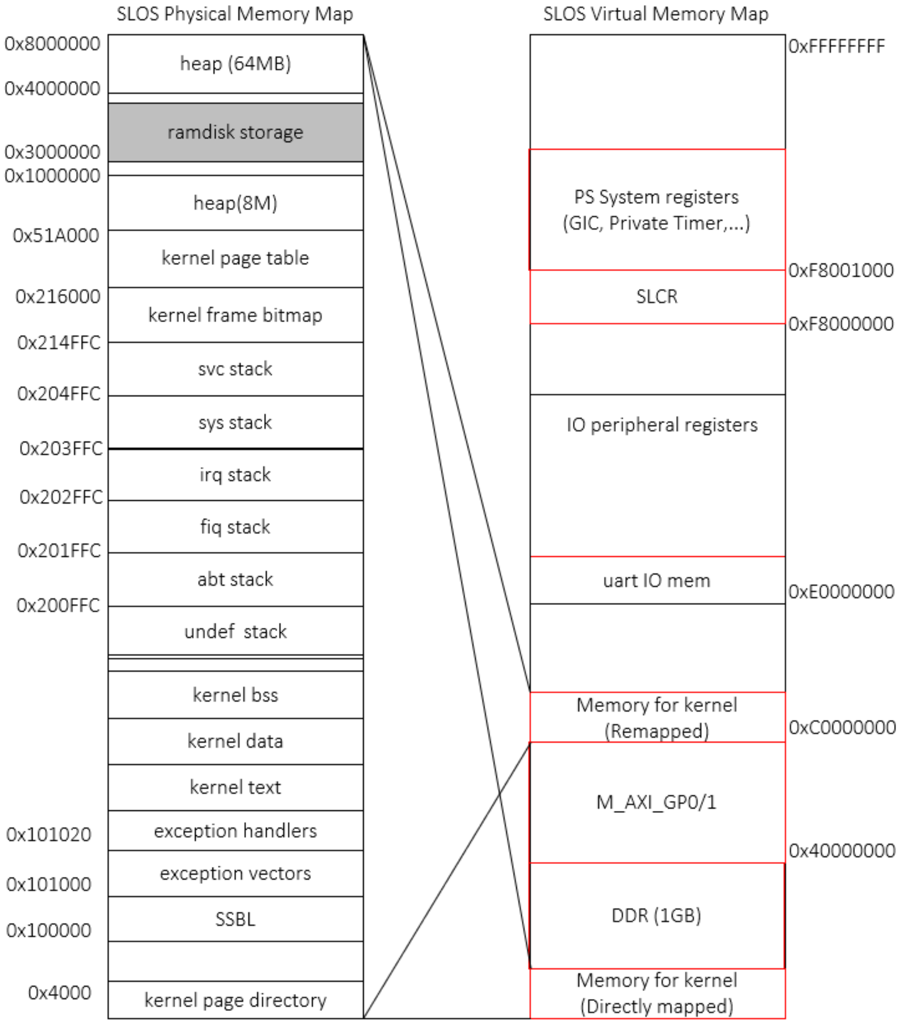


Figure 6-5: Modified memory map for ramdisk storage

6.2.4 SLFS Blocks Layout

After mounted and formated, SLFS file system composes the ramdisk region with *superblock, inode bitmap, data block bitmap, inode table* and *data block*. The layout of SLFS in the disk looks like below figure 6-6. The size of each meta data block is 4KB. The total size of the storage disk is limited to 4MB. As mentioned, ramdisk is used for storage media in this book. Practically, the ramdisk size in SLOS is so small that it can't store many files. But for our hobby project, this capacity of disk is enough because we will store only one simple user application into this storage.

Superblock is a meta data of a file system itself. It is a record of the file system characteristics including its size, block size, usage information, location of inode table and so on. If this block is broken, the entire file system can't be mounted and the whole data gets lost. So, some file system has a copy of this superblock for recovery. Nonetheless, 4KB superblock in SLFS doesn't have any meaningful information. It is just a blank 4KB block in ramdisk.

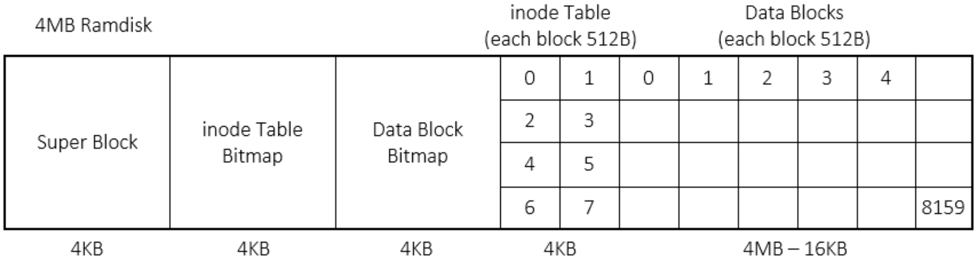


Figure 6-6: SLFS file system layout

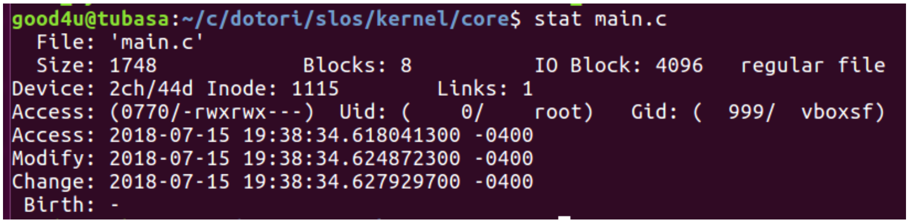
Each inode table bitmap and data block bitmap is the bitmap showing the allocation status of inode table and data blocks respectively. One bit of the bitmap represents the occupation of a block in the inode table or in data blocks. Inode table is a table containing the inode descriptor for an associated file. The size of inode in SLOS is fixed as 512bytes. One 4KB inode table bitmap can cover 4096 * 8 entries, but SLFS just support only one 4KB size inode table which has only 8 inode entries.

The size of data block is also 512bytes. Since each block size in a ramdisk is 512B, 4KB data block bitmap can cover up to 16MB (4096 * 8 * 512B) data block. Data block is an array of 512byte blocks that contains the byte data of a file. SLOS's ramdisk is only 4MB size and there are 4 * 4KB blocks are reserved. The total data block number is (4MB - 16KB) / 512B = 8160 blocks.

6.2.5 Inode for SLFS

The inode is a data structure in a *Unix* style file system that describes a file system objects such as a file or a directory. Each inode stores the attributes and disk block locations of the object's data [1]. Inode is a unique serial index containing important information for a file. Inode is a meta data of a file. It contains all information that a file system needs to manipulate its operations for a file. The *stat* command in Linux can list up some information in inode as in figure 6-7. The inode contains

1) file size in byte
2) Device represented by the inode
3) number of blocks allocated to this file
4) I/O block size
5) User, Group ID for this file
6) A link count telling how many hard links points to the inode
7) Pointers to the data blocks



```
good4u@tubasa:~/c/dotori/slos/kernel/core$ stat main.c
  File: 'main.c'
  Size: 1748            Blocks: 8          IO Block: 4096   regular file
Device: 2ch/44d Inode: 1115        Links: 1
Access: (0770/-rwxrwx---)  Uid: (    0/    root)   Gid: (  999/  vboxsf)
Access: 2018-07-15 19:38:34.618041300 -0400
Modify: 2018-07-15 19:38:34.624872300 -0400
Change: 2018-07-15 19:38:34.627929700 -0400
 Birth: -
```

Figure 6-7: Linux *stat* command to show the inode

The example in figure 6-7 shows the basic meta data of a file. The file size is 1748 bytes but the allocated block number is 8 blocks. If you want to know one block size in the file system, a *stat -c %B main.c* command can be used. It shows 512 bytes of a block size in my case. Then, the file I/O device aggregates 8 blocks for one I/O transaction and this is why the file smaller than 4096 bytes is allocated 8 data blocks. You also can see the *device ID, inode number, access permission,* and so on with the *stat* command.

Notice the *pointers to the data blocks* in the list for inode. The data blocks are the blocks in the storage to save the data of a file. Inode has a pointer to this data blocks. In SLFS, this is the only information that inode needs to have. The *struct inode* structure in SLFS looks like below. It is very simple. The *struct inode* structure has only inode number, file size and array to point the location of data blocks.

```
struct inode {
        uint32_t iNum;
        uint32_t file_size;
        uint32_t blkloc[INODEBLKMAX];
};
```

The block location index is the block number where the file data is placed in the data block region. As you can see the SLFS file system layout in figure 6-6, there are only 8160 data blocks and each block size is 512B in SLFS. The *blkloc[]* array has an index to these block data. This array size is limited to *INODEBLKMAX* which is 126. This is because one inode itself has a 512B size and there are two *uint32_t* members, then the blkloc array can have up to 126 entries. The relation between inode *blkloc* index and data block location is depicted as below figure 6-8. The file read/write operations need to handle the *blkloc[]* array in inode and the data blocks.

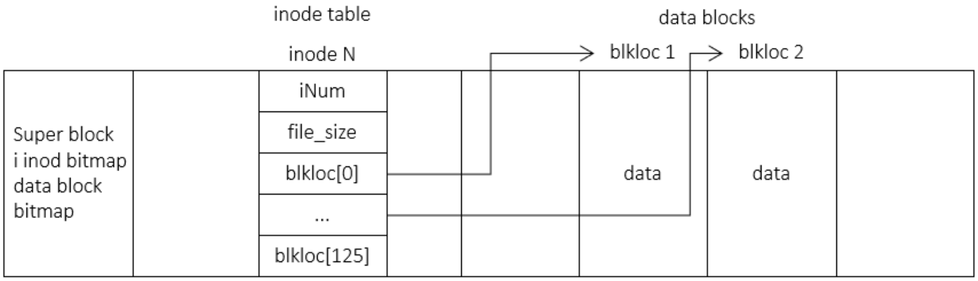


Figure 6-8: Relations between inode, blkloc array and data blocks.

6.2.6 File System Initialization, Mount and Fomat

SLFS is also needed to be mounted and formatted before being used. Mount in SLFS is simply initializing the parameters of file system structure. There is a *struct file_system* for SLfS and it looks like below.

```
struct file_system {
     uint8_t *pIBmp;
     uint8_t *pDBmp;
     uint8_t *SuperBlkStart;
     uint32_t SuperBlkStartBlk;
     uint8_t *inodeBmpStart;
     uint32_t inodeBmpStartBlk;
     uint8_t *DataBmpStart;
     uint32_t DataBmpStartBlk;
     uint8_t *inodeTableStart;
     uint32_t inodeTableStartBlk;
```

```
        uint32_t inodeTableSize;
        uint8_t *DataBlkStart;
        uint32_t DataBlkStartBlk;
        uint32_t bMounted;
        uint32_t fdCnt;
        struct file *fpList[INODE_NUM];
        uint32_t BlkSize;
    };
```

One-time member variables of this struct are initialized in init_file_system() function and variables related to file system layout are initialized in mount_file_system() function. init_file_system() function looks like below.

```
1     struct file_system *init_file_system()
2     {
3         int i;
4         pfs = (struct file_system *)kmalloc(sizeof(struct file_system));
5
6         pfs->bMounted = 0;
7         pfs->inodeTableSize = INODE_NUM;
8         pfs->BlkSize = DATA_BLK_SIZE;
9         pfs->pIBmp = pfs->pDBmp = 0;
10        pfs->fdCnt = 0;
11
12        for (i = 0; i < INODE_TABLE_SIZE; i++) {
13            pfs->fpList[i] = NULL;
14        }
15        return pfs;
16    }
```

The init_file_system() function allocates the struct file_system structure in the heap in line 4. Then, it initializes the file system member variables (Remember the demanding page happens here). Notice that the line 12 ~ 14 initializes the file pointer (struct file *) list in the file system. The struct file_system variable has almost all information about the file system.

mount_file_system() function looks like below.

```
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->pIBmp = (unsigned char *)(RAMDISK_START + INODE_BITMAP_START);
             pfs->pDBmp = (unsigned char *)(RAMDISK_START +
10           DATA_BLK_BITMAP_START);
11
12           pfs->bMounted = 1;
13   }
```

pfs is a pointer to the file system structure variable that is allocated in the init_file_system(). The pfs has pointers (SuperBlkStartBlk, inodeBmpStartBlk, DataBmpStartBlk, inodeTableStartBlk, DataBlkStartBlk) to each region of the file system layout in figure 6-6, and can access each region in the file system. As you can see the lines 3 ~ 10, the mount_file_system() function initializes those region pointers with predefined values. The block offset values are defined in *kernel/inc/file_system.h.*

Format SLFS is done in format_file_system() and it wipes out all data in the ramdisk. format_file_system() erases the Superblock, inode bitmap, data block bitmap, inode table and all data contents in the data blocks.

6.2.7 Two Level Data Block Indexing

Inode in SLSF file system has an index array to locate its data blocks. Since inode size is 512bytes, this array index size is limited to 504 (512 - sizeof(uint32_t) * 2) byte that has 126 index array entries. This means the maximum data size that one file can have is 63KB (126 * 512 byte). Even we pursue a simple and light, this max file size seems too small for a file. To increase this max file size, we have to increase the size of data block index array in inode. We can achieve this simply by adding one more layer between this blkloc array and the data block. The first data block indexed by inode’s blkloc array doesn’t have any data for the file. Instead, it has an index for another block location where the real file data are placed. This is a second level block location index array. The two-level data block indexing looks like figure 6-9.

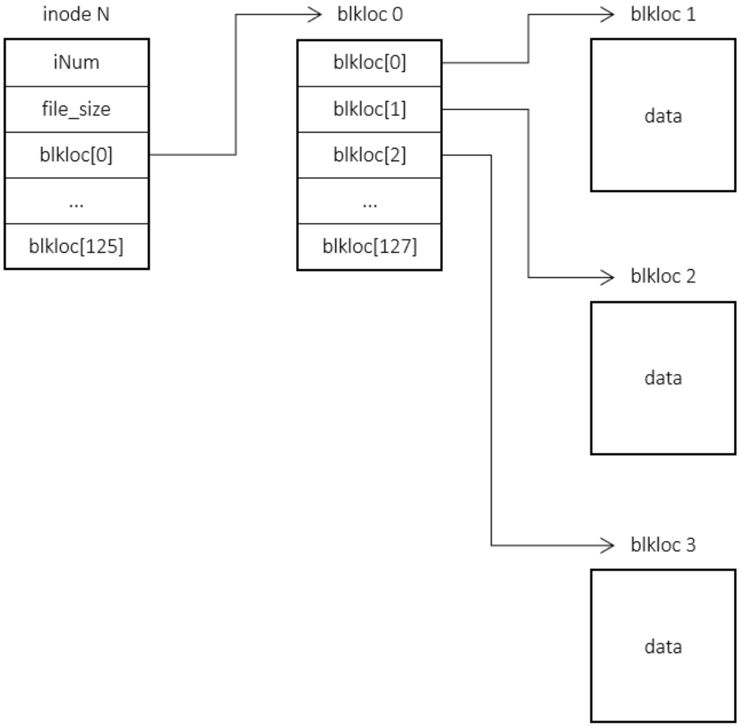


Figure 6-9: Two level data block indexing of inode

In figure 6-9, the inode N for file N has two level data block indexing. The blkloc[0] which is in the inode N contains the first-level data block index, but in this case, the data block doesn't have any data for a file. Instead, the first-level data blocks have a special data that is second-level data block's index. This data block is a two-level block location indexing. Since the data block size is 512bytes, the second-level block locations in each data block are up to 128 entries. By adding the second-layer between real data block and inode blkloc array, the max data size of one file can have is increased up to about 8MB (126 entries * 128 entries* 512byte). This file size is big enough for SLFS which has only 4MB ramdisk.

6.3 File Operations in SLFS

File is the final instance of the file system. Data in the storage hardware is accessed through the file. SLFS has also files. Inode in SLFS is a meta data for a file and is associated with a file: one file is associated with one inode. File in SLFS is nothing but an inode in the inode table. SLOS's file operations such as *open, close, read, write* don't work together with system calls as a normal operating system does. SLFS has a *struct file* structure for a file descriptor. It looks like below.

```
1   struct file {
2       struct file_system *pfs;
3       uint32_t fd;
4       uint32_t pos;
5       uint32_t fsz;
6       uint32_t oCnt;
7       char name[128];
8   };
```

The line 2, struct file_system pointer is for linking the file with the file system. Line 3 ~ 6 are number values to describe the file such as file descriptor number, file pointer position, file size and file open count. The name in line 7 is for file name.

6.3.1 Ramdisk IO Driver

Read and write file operations need a device driver functions that physically read and write to the storage device. This device driver virtualizes the physical layer to the logical file operations. SLFS has only ramdisk for its storage and the device driver for disk IO is just a memory read and write. These are simply memory load (read) and store (write) instructions. This saves us time to work on device driver for storage media and makes it pretty easy to develop logical functions. Nonetheless, SLFS has a 512 byte of data block size, one transaction of read and write should be done by this size. In other words, even writing 1 byte of data to a file occupies 1 block of data. This results in the internal fragmentation. Ramdisk I/O device driver can be implemented by memory load/store with the size of 512 byte as below.

```
void read_ramdisk(int mem_blk_num, char *buf)
{
    int i;

    for (i = 0; i < DATA_BLK_SIZE; i++) {
        buf[i] = ((char *)(RAMDISK_START + mem_blk_num *
        DATA_BLK_SIZE))[i];
    }
}

void write_ramdisk(int mem_blk_num, char *buf)
{
```

```
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      }
```

This device driver looks quite simple. read_ramdisk() and write_ramdisk() get the block index number and a data buffer. Line 5~7, 14~15 show that read/write data buffer to storage is simply done by normal memory access. Notice that a single read/write operation is done with the size of 512 bytes. These functions get a 512bytes data buffer and read one data block from ramdisk to the buffer, or write the data buffer to the one data block of ramdisk. If we use an HDD or other storage, the device driver for those media should be implemented. As in figure 6-5, SLOS allocates the 4MB memory from 0x0300_0000 for the ramdisk storage. The ramdisk I/O driver manipulates read/write operations for this region of a storage disk.

6.3.2 File Open, Close and Delete

File open has a string parameter for the file name. This file name should be unique. Since there is no directory (or path to the file) in SLFS, the file name is the only way to distinguish each file. 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 matched name, then file open function will go through the steps to create a new file and return the file pointer of that. Below is the open_file() function implementation to open a file in SLFS.

```
struct file *open_file(char *str)
{
    int _fd;
    struct file *fp;

    fp = find_file_by_name(str);
    if (fp != NULL) {
        fp->oCnt++;
        return fp;
    }

12          fp = (struct file *)kmalloc(sizeof(struct file));
13          _fd = register_file(fp);
14          if (_fd < 0) {
15              kfree((uint32_t)fp);
16              return NULL;
17          }
18
19          fp->pfs = pfs;
20          fp->fd = _fd;
21          fp->pos = 0;
22          fp->fsz = 0;
23          strcpy(fp->name, str);
24          file_system_create_file(fp);
25          fp->oCnt += 1;
26          return fp;
27      }
```

str is a pointer to a string of the file name. This function returns the file pointer to the old file descriptor if there is already a file with the same name as pointed by *str*. Otherwise, it creates a new file and returns a file pointer to that new file descriptor. Opening a file in SLFS is just a registration of a new inode to the inode table. Line 6 ~ 10 looks for the file with the same name string. If there is already that file, this function increases the *oCnt (open count)* and returns the file pointer of that. If there is not such a file, open_file() allocates a file descriptor from the heap (line 12) and registers an inode for that new file to the inode table (line 24). While doing these, it does some initialization of file data structure such as file size, file offset position pointer and increase the file open counter by one (line 19 ~ 23). After inode is registered, all R/W for that file is manipulated via this inode.

The file_system_create_file() function in line 24 creates all meta data related to current file in the file system. It gets the file descriptor allocated from line 12. It does almost all about the file creation. This function is defined in *kernel/core/file_system.c.* Let's look into it more detail.

```
1       int32_t file_system_create_file(struct file *fp)
2       {
3           int i, blk_off, byte, bit, inodeIdx;
4           struct inode *pinodeDat;
5           char blk_data[DATA_BLK_SIZE] = {0,};
6
7            inodeIdx = fp->fd;
8            if (inodeIdx >= INODE_NUM) {
9                return 1;
10           }
11           blk_off = (int)((inodeIdx >> 3) / pfs->BlkSize);
12           byte = (inodeIdx >> 3) % pfs->BlkSize;
13           bit = inodeIdx % 8;
14
15           read_ramdisk(pfs->inodeBmpStartBlk + blk_off, blk_data);
16           if ((blk_data[byte] & (0x1 << bit)) != 0x00) {
17               return 2;
18           }
19
20           blk_data[byte] |= (0x01 << bit);
21           write_ramdisk(pfs->inodeBmpStartBlk + blk_off, blk_data);
22
23           for (i = 0; i < DATA_BLK_SIZE; i++) {
24               blk_data[i] = 0x00;
25           }
26           pinodeDat = (struct inode *)(blk_data);
27           pinodeDat->iNum = inodeIdx;
28           pinodeDat->file_size = 0;
29
30           /* write inode data for new file to inode table */
31           write_ramdisk(pfs->inodeTableStartBlk + inodeIdx, blk_data);
32
33           return 0;
34       }
```

Line 11 ~ 18 finds out the location in the inode bitmap and determines whether the file already exists or not. This was already checked in the open_file() but it is checked here again. Line 15 reads the bitmap data by using the read_ramdisk() function. If there is not the file, file_system_create_file() function allocates a new inode in the inode bitmap in line 20 ~ 21. Line 26 ~ 28 initializes the inode data for the new file and line 31 writes the inode data into the associated inode block in the inode table. After file_system_create_file() function returns, there is a new inode associated with the current file descriptor. A new file creation means a creation of a new inode.

Closing a file in SLFS is quite simple, decreases the file open counter of that file. Even the file open count is zero or negative, SLFS doesn't care about those cases. close_file() function is as below.

```
uint32_t close_file(struct file *fp)
{
        fp->oCnt--;
        return 0;
}
```

Instead, in order to remove a 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 from heap. So, the SLFS still keeps the files in the ramdisk after those files are closed and allows other applications to access that file with the file name again. delete_file() function deletes the file and remove the inode. Below is the implementation of delete_file() function.

```
uint32_t delete_file(struct file *fp)
{
        unregister_file(fp);
        file_system_delete_file(fp);
        release_blks(fp);
        kfree((uint32_t)fp);
        return 0;
}
```

delete_file() function gets a file pointer as its paramenter and calls unregister_file(), file_system_delete_file(), release_blks() and finally kfree() to release the file descriptor in kernel heap.

6.3.3 File Read

File read function accesses to the data block and read data blocks to the application buffer. As discussed in chapter 6.2.5, the inode has the blkloc array which is the index of the data block allocated to that file. The read function's declaration looks like below.

1) *uint32_t read(struct file *fp, uint32_t _n, char *_buf)*
   fp: a file pointer for reading

_n: data byte amount to read
_buf: a pointer to a buffer to which block data should be copied
return value: the data amount read

The read() function should take care of two things. First, while reading, the read() function should consider a two-level of data block indexing. In other words, the data block indexed by blkloc array in inode has another index for the second level of data block. The second-level data block has the real data for being read. Second, since every read is done for the amount of 512bytes, the read() function should consider current file position and the remainder of data block. The read() function carefully handles the data crossing over the multiple data blocks. File read follows below flow in figure 6-10.

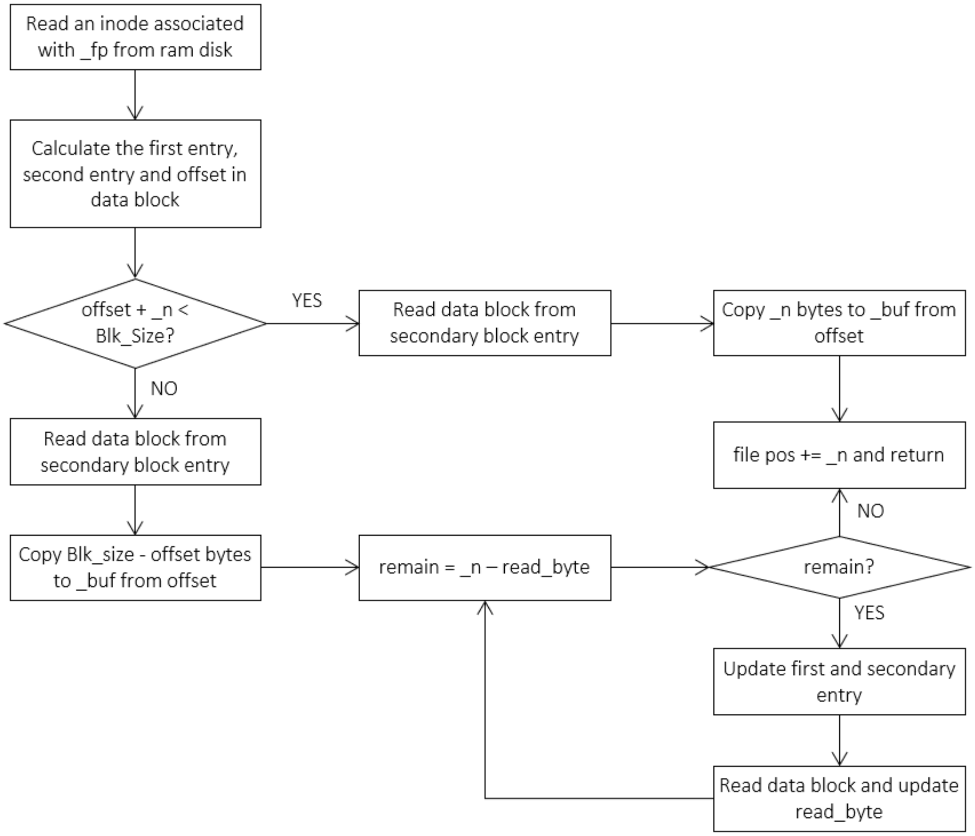

Read an inode associated with _fp from ram disk
Calculate the first entry, second entry and offset in data block
offset + _n < Blk_Size?
YES
NO
Read data block from secondary block entry
Copy _n bytes to _buf from offset
file pos += _n and return
Read data block from secondary block entry
Copy Blk_size - offset bytes to _buf from offset
remain = _n – read_byte
remain?
NO
YES
Update first and secondary entry
Read data block and update read_byte

Figure 6-10: File read() function flow

File read function first calculates the current file offset and entries of the first and second-level data block index. Then, since the ramdisk read operation happens only with 512 bytes, the read() function checks whether current file read can be done within this block or not. If it is possible to read whole data within current block, read() function traverses the index blkloc entries and reads the data block to out buffer then, returns to the caller. If this is not possible, then read() function repeats reading the data blocks until the remaining bytes becomes zero. After whole data are read, the read() function updates its file position and return. We will not cover this function line by line. The read() function implementation is in *kernel/core/file.c.*

6.3.4 File Write

The write() function accesses data blocks and writes the content of input buffer into the data blocks in ramdisk. This function uses inode to access data block region. Unlike read() function, write() function needs to find out free data blocks and to update the data block bitmap. Other implementations are similar with the read() function such as it needs to consider the 512B data block boundary. The write() function follows below flow in figure 6-11.

Below is the write() function declaration.

1) *uint32_t write(struct file *fp, uint32_t _n, char *_buf)*
   fp: a file pointer for reading
   _n: data byte amount to write
   _buf: a pointer to the buffer from which data should be copied
   return value: the written data amount

The write() function need to find a free data block. The find_datablk() function in the file_system.c does this. It traverses the data block bitmap and finds a free data block then returns the index of free data block. Then, the write() function follows similar steps as the read() function; finds a new data block and write until remaining data amount becomes zero then update the file size, inode blkloc array and returns the byte amount of data written. The detailed description on the implementation of write() function is skipped. The write() function implementation is in *kernel/core/file.c.*

6.3.5 File Read/Write Test

To test the file system implementation, let’s run a test 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() function in main.c. As we did before, the start_kernel() function is the place to put the SLOS’s kernel initialization routines. After start_kernel() function finishes the initialization of the memory

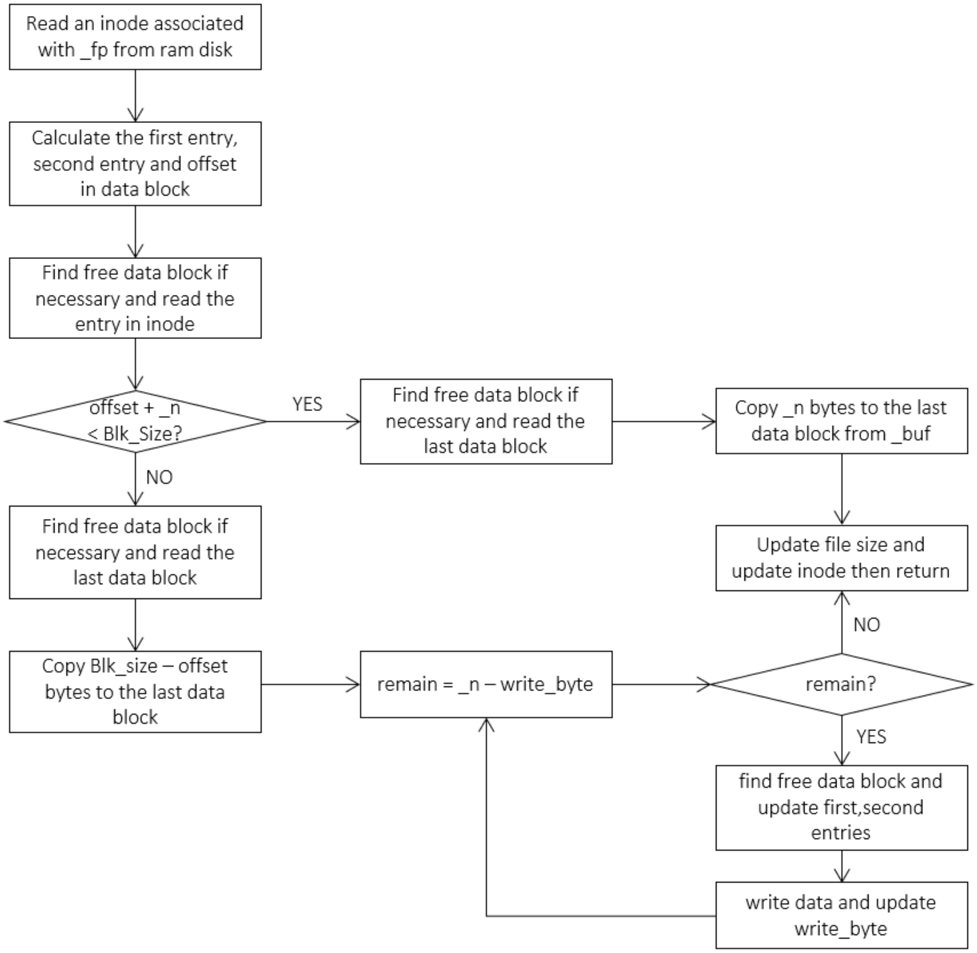

Read an inode associated with _fp from ram disk
Calculate the first entry, second entry and offset in data block
Find free data block if necessary and read the entry in inode
offset + _n < Blk_Size?
YES
Find free data block if necessary and read the last data block
Copy _n bytes to the last data block from _buf
NO
Find free data block if necessary and read the last data block
Update file size and update inode then return
NO
Copy Blk_size – offset bytes to the last data block
remain = _n – write_byte
remain?
YES
find free data block and update first,second entries
write data and update write_byte

Figure 6-11: File write() function flow

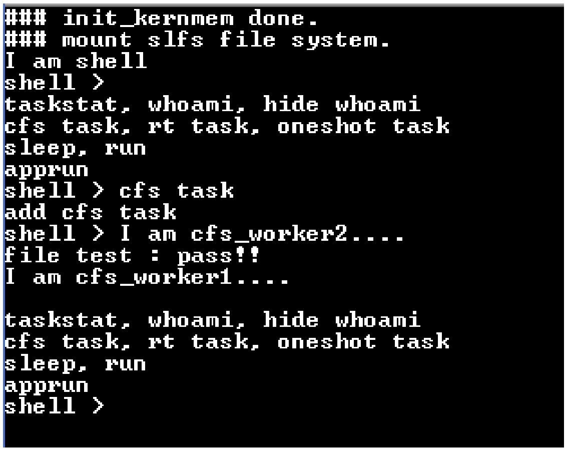
management from chapter 5, the gic controller and the idle task initializations, the start_kernel() runs the initialization routine for SLFS file system. The start_kernel() from chapter 5 is now changed as follows.

```
int start_kernel(void)
{
    struct framepool framepool;
    struct pagetable pgt;
    struct vmpool kheap;

    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 newly added for SLFS. These lines are for initialization of SLFS file system. It calls init_file_system(), mount_file_system(), and format_file_system() sequentially. After that the process management initialization follows.

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

```
fp = open_file("test");
for (i = 0; i < FILE_TEST_LEN; i++)
                buf[i] = i % 256;
write(fp, FILE_TEST_LEN, buf);
reset(fp);
read(fp, FILE_TEST_LEN, temp);
xil_printf("file test : ");
for (i = 0; i < FILE_TEST_LEN; i++) {
                if (buf[i] != temp[i]) {
                    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;
```



```
### init_kernmem done.
### mount slfs file system.
I am shell
shell >
taskstat, whoami, hide whoami
cfs task, rt task, oneshot task
sleep, run
apprun
shell > cfs task
add cfs task
shell > I am cfs_worker2....
file test : pass!!
I am cfs_worker1....

taskstat, whoami, hide whoami
cfs task, rt task, oneshot task
sleep, run
apprun
shell >
```

Figure 6-12: File Write and Read test in cfs_worker task

Line 1 opens a new file with the name of *'test'*. This file name should be unique for SLFS file system. open_file() function checks whether that file already exists or not. If it doesn't exist, SLFS creates a new file in file_system_create_file() function then returns a file pointer of that file.

Line 2 writes a character data buffer to the test file. Be careful the *buf* array size is not too big because it is declared as a local variable. Since every task in SLOS has 4KB stack size, if this buf array is too big, that could result in a *stack overflow*. The stack overflow could bring about the kernel crash. If you want to use bigger buffer to test the file, then allocate this array in the heap memory by using kmalloc().

Line 5 resets the file pointer position (the *pos* variable) to the start location of the file for the next file read operation.

In line 6, file read operation saves the read data to temp buffer. The write() and read()

functions write and read data by the amount of FILE_TEST_LEN. Currently, this size is set as two data block size which is 1024bytes.

The rest line 8 ~ 18 verifies the file read and 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 in the file is printed. Figure 6-12 is for the SLFS file write and read test by adding the test routine to the one of the CFS task.

Then line 20 ~ 21 close that file. Even after close this file, that file still remains in the file system. You can still access that file with open_file() later. The delete_file() function will remove the file permanently.

6.4 System Call

System call is an interface between user application and operating system. We are going to implement only one system call in this chapter for "helloworld" user application to access the system hardware resource. From this chapter, we will demonstrate how user application is loaded into an SLFS file system and how it is working with SLOS through the system call.

6.4.1 System Call - How it Works

The applications in high level operating system such as Linux or MS Windows can run in different systems with different hardware configurations. Since these user-built applications have no idea about the system hardware, the operating system should expose the interface of the system hardware resources to these applications. For demonstration, the applications don't have any idea on how to allocate heap memory for its use or don't know how to save its data to the persistent storage. A system call in the operating system allows these user applications to access those hardware resources. System call is a middle layer between applications and operating system.

Second thing to keep in mind for system call is the processor modes, privilege mode and non-privilege mode. Recall chapter 2.5.7, ARM Privilege Levels. ARM processor has privilege levels for its runtime modes and each privilege level has its own scope to access system resources. For example, *Supervisor mode (SVC)* of processor is in *PL1* privilege level, which can access the whole system resources. So, the operating system runs in this mode. But the user applications run in *User mode (USR)* which is in *PL0* and this mode can't access the system hardware resources. Nonetheless, many user applications still need the service of system hardware, for example saving its data into a file in the storage hardware. System call is used for this purpose: expose system hardware from privileged mode to user applications running in unprivilged mode.

Normally, the system calls are not invoked directly, rather they are used through wrapper functions. Operating system should implement the system calls for each predefined system call number. This system call number is delivered to operating system through the system call wrapper library. In order to use the system's privileged resources, user application has to link the wrapper library and calls the system call wrapper functions. A famous example of this system call library is a *glibc* library *(GNU C Library)*. There are hundreds of system call implementations in Linux for *glibc* library such as *open(), read(), write(), fork(), exec(), malloc()* and so on.

Generally, context switching doesn't have to occur when the system call of kernel implementation is invoked from user application. System call can be achieved only by changing the processor's mode from user mode to supervisor mode. User application can deliver some data to the kernel and kernel also can return the result to the user applications. As in table 2-5, the user mode and other privileged modes of ARM processor share general purpose registers from r0 to r7. In this case, kernel services the system calls on behalf of user application. User application and kernel are combined together and work as one user application running. But sometimes kernel scheduler still needs to switch out the user application while service a system call. This happens when the system call is a blocking system call such as waiting the I/O transaction to be done. This blocking I/O makes the kernel's scheduler put the user application into *waitqueue* and schedule in next task for not losing the cpu time. A simple usecase of system call, malloc() is depicted in figure 6-13. This is non-context switching case and user application plus kernel works as a single application. This figure implies there are many in-and-outs between user mode and kernel mode while an application is running.

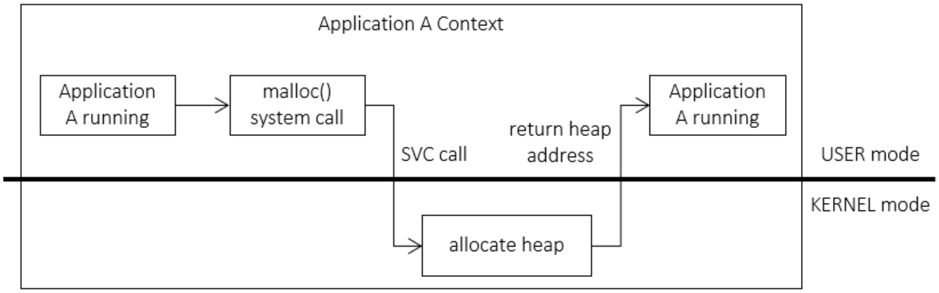


Figure 6-13: Application combined with kernel's system call (Non-context switching case)

6.4.2 SVC Handler Implementation

There is an ARM ISA instruction for the system call implementation. The *'SVC' (Supervisor call)* instruction or *'SWI' (Software Interrupt)* instruction can be used for system call implementation. These instructions are used for the application to delegate the privileged operations such as accesses to system hardware resources to the operating system. When these instructions run, the processor goes into exception state and the system call handler at offset 0x0000_0008 from vector base is called. Now, it's time to fill out another exception handler - the system call handler. After this chapter, we will have reset vector handler at offset 0x0, system call handler at offset 0x8, data abort handler at offset 0x10, and irq handler at offset 0x18.

Normally, there are hundreds of different system calls. Each of these system call has a unique number. This number indicates the type of system call that the caller is requesting. The system call number must be transferred to the kernel. But, SLOS has only one system call, a *'write'* system call, to export the UART hardware resource for an application to print a string message into terminal window. The *write* system call number in SLOS is 2. This system call number is embedded into the SVC system call instruction.

SLOS has another directory that stores the system call library implementation. This directory is separated from the kernel directory which is the top-level directory for all SLOS's kernel sources so far. A *libslos* is a new directory holding the files of system call library and syscall.S, print_mesg.c in libslos directory are the files having write system call implementation. The write system call implementation in syscall.S is very simple as below.

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

Since there is no context swithcing in system call sequence, running a system call is same as a normal function invocation. A little difference in this case is that the function needs to run a special ARM instruction which is a *Supervisor Call (svc)* instruction. Running a svc instruction in line 3 is all for system call implementation in the library. The svc instruction, previously known as *Software Interrupt (swi),* causes a supervisor call exception. This exception makes the processor go into a *Supervisor mode* that can run privileged instructions. A 32bit ARM svc instruction is encoded as below figure 6-14.

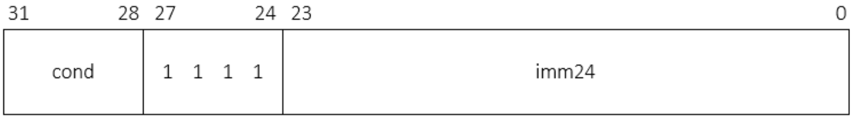


Figure 6-14: *SVC* instruction format

The 24bit immediate constant field, *imm24* (bit[23:0]), holds the value of system call number. The write function embeds decimal 2 to this imm24 field in the svc instruction. The kernel system call handler parses this instruction and extracts 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 sets the address 0xC010_1000 as its VBAR, the syscall exception handler address must be 0xC010_1008. 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}
```

Notice line 4 that loading the encoded svc instruction. Line 4 loads the svc instruction into scratch register r12. Since svc instruction return address is lr which is the next instruction of svc*,* the encoded svc instruction can be simply accessed by using address lr - 4. After loading the svc instruction into r12*,* getting the system call number is easy. Line 5 ~6 is gets the imm24 bits for the system call number by masking out the upper 8bits and put the system call number into r2. The register r1 holds the argument directly coming from user application. After filling out the registers for system call arguments, the system call handler calls a platform system call handler. The platform_syscall_handler() declaration is below.

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

The first parameter is an argument pointer from user application. The application can deliver user space data into kernel through this pointer address. In the *write* system call, this pointer points to the buffer storing the application's message string. The string buffer can be associated with this pointer parameter between user space and kernel space. 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.

```
int platform_syscall_handler(char *msg, int idx, int sys_num)
{
    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. Notice that the message buffer address is recalcuated as line 7. Normally, when user application is compiled, it is assumed that user application starts at address 0x0. Later, the *ELF loader* in SLOS loads the user application into a specific memory region dedicated for user applications. The start address of this region is defined as *USER_APP_BASE*. So, when user application is running, the symbol address has the offset of the *USER_APP_BASE* amount. In addition, the user applications are loaded with a predefined offset from the base address. This offset gap is defined as *USER_APP_GAP*. The descriptions on how user applications are loaded into memory map will be handled again later. So, the string buffer address delivered from user application is based on 0x0 base address. The new address while the application running is recalculated as line 7 by taking into account the offset values.

The 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 instruction like this way. But the SLOS doesn’t have a USR mode, which means all user applications also run in SVC privileged mode. For proper operation, user applications should run in USR mode and when system call is invoked, the system call handler change the processor mode to SVC mode and process the privileged operations and return to user application with restoring the USR mode.

6.4.3 Syscall Library for User Application

User application is built separately with SLOS but has to access SLOS functionalities. To bridge user application and SLOS kernel, there is a *libslos.a* library. User application needs to link this library while it is being built. This library is composed of two files; print_mesg.c and syscall.S in libslos directory.

print_mesg.c has print_mesg() function which is supposed to be called by user application. The print_mesg() function implementation is below.

```
int print_mesg(const char *buf, const int idx)
{
        write(buf, idx);
        return 0;
}
```

The print_mesg() function is very simple. It calls only the write() function which is placed in syscall.S. The write() function is covered in previous chapter 6.4.2, SVC handler implementation. It just runs a supervisor call (svc) instruction with the argument of string buffer address and task index number. The task index number is used to calculate the offset of the user application from the *USER_APP_BASE.*

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

libslos.a file 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 libslos.a, add below changes to the Makefile. The variable *$(AR)* should point the arm-none-eabi-ar.

```
$(OUT_TOP)/libslos/libslos.a:$(OUT_TOP)/libslos/syscall.o $(OUT_TOP)/libslos/print_mesg.o
        $(AR) rc $@ $^
```

This command in Makefile will archive the syscall.o and print_mesg.o object files to libslos.a library. The user application should link this library statically to access the system calls.

6.5 HelloWorld Application and Ramdisk

The C compiler is able to generate code from many separately compiled modules. For programs to execute successfully these modules must be able to interoperate, with the operating system code and any code that is written in assembler or any other compiled language. For that reason, we must define a set of conventions to govern inter-operability between separate pieces of code. The *Application Binary Interface (ABI)* for the ARM architecture specification describes a set of rules that an ARM executable must adhere to in order to execute in a specific environment. It specifies conventions for executables, including file formats and ensures that objects from different compilers or assemblers can be linked together successfully. There are variants of the ABI for specific purposes, for example, the Linux ABI for the ARM architecture or the Embedded ABI (EABI) [5].

Now, we have a system call library, it's time to build a user application. Our compiler tool will put all this together to make it run in the processor. This application is going to be used to test a SLFS and a system call. A ramdisk.img is created to store user applications. The ramdisk.img file is not recognized as a file in SLOS, rather it has compiled binary codes. Its content will be loaded into the SLFS later and loaded into SLOS memory map by *ELF loader*.

6.5.1 HelloWorld Application

All things that helloworld user application does is to print a string message into the terminal window. This application accesses the system hardware via the write system call that is exposed by the libslos.a library. The helloworld application simply looks as below.

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

print_mesg() function is implemented in *libslos/print_mesg.c* and compiled into libslos.a. Since there is no exit() system call in SLOS, this user application should spin forever at the end of its body. Since there is no dynamic loader in SLOS, the object file of helloworld application should be statically linked with the libslos.a. To build this helloworld application, below build process is added to Makefile.

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

The -*static* option in the last line means the output helloworld executable binary is statically linked with libslos.a library. The -*lslos option* defines the library name. With these steps, the helloworld binary is linked with the system call library and can print its message to the terminal.

6.5.2 Building ramdisk.img

ramdisk.img is a binary file containing all user application executables. There is a *mkfs* tool to build ramdisk.img. mkfs is a standalone application tool. The source file is located in *mkfs/mkfs.cpp*. The mkfs tool reads the compiled user application binary files and appends it to the end of ramdisk.img.

The first 4 bytes in ramdisk.img is the number of user applications embedded into the ramdisk.img. Next, there is 4 bytes to hold the size of each user application. The user application must be padded to make 4byte alignment. mkfs tool also pads the user application for 4byte alignment. If there is padded bytes, the size value is updated with the padding bytes before written into the ramdisk.img. Then, the user application binaries 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 layout is built simply as below.

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

Figure 6-15: ramdisk.img file layout

The mkfs tool gets file name in the argument and then reads the file, pads it if necessary, and append it to the ramdisk.img. The tool just concatenates every application binary. Its implementation is quite simple and explanation is skipped. Makefile also need to be changed to build this tool. This tool must be built before creating the ramdisk image. Refer the *mkfs/mkfs.cpp* file for better understanding.

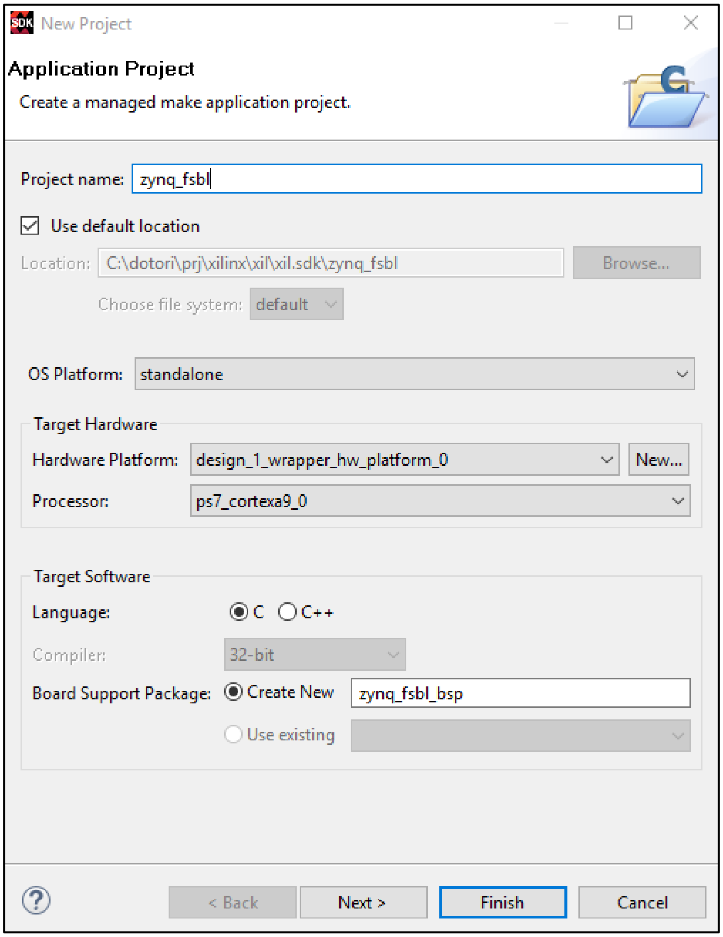
6.5.3 Loading ramdisk.img from fsbl

In Linux, the final OS image is built by merging the kernel image and init ramdisk image. For example, the final linux image contains the kernel image combined with *root* file system image. The root file system has useful tools and initial services which run after kernel booting is completed. After the bootloader loads the OS binaries into corresponding address, the kernel boot starts first. Then, after kernel boot is done, kernel mounts the root file system and runs an init process.

But in SLOS, the ramdisk.img is a separate file from kernel.elf. The Xilinx FSBL bootloader loads only kernel.elf. So, we have to tweak a current Xilinx FSBL bootloader (zynq_fsbl.elf) to read the ramdisk.img from SD card and load it to a specific memory location. If you don’t want to change the FSBL source files, there is a prebuilt FSBL bootloader that loads the ramdisk.img file into memory. You can download the prebuilt bootloader from the tools repository. Run below command to clone the tools repository.

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

Then, you can see the zynq_fsbl_hook.elf file in the tools directory. This file is a modified FSBL bootloader that load the ramdisk.img to a specific memory location. You need to include this to build the BOOT.bin instead of including the previous zynq_fsbl.elf.



New Project

Application Project

Create a managed make application project.

Project name: zynq_fsbl

☑ Use default location

Location: C:\dotori\prj\xilinx\xil\xil.sdk\zynq_fsbl

Choose file system: default

OS Platform: standalone

Target Hardware

Hardware Platform: design_1_wrapper_hw_platform_0

Processor: ps7_cortexa9_0

Target Software

Language: ● C ○ C++

Compiler: 32-bit

Board Support Package: ● Create New zynq_fsbl_bsp

○ Use existing

Figure 6-16: Application Project setup

If you are still interested in modifying the Xilinx FSBL source code, follow below steps. We need to use the Zynq default block design in chapter 2.3.7. After finishing bitstream generation of the Zynq default block design in Vivado, go to File -> Launch SDK and click OK. Then, Vivado will launch an *eclipse IDE*. The eclipse automatically creates the hardware description wrapper project. In eclipse window, go to File -> new -> Application Project. Set the Application Project window as in figure 6-16. Set the project setup as in figure 6-16. Then, click Next button. Don’t click Finish. We will see the Template window. Choose ‘Zynq_FSBL’ in the available templates and click Finish button. The eclipse and Xilinx SDK will automatically create ‘zynq_fsbl’ project and ‘zynq_fsbl_bsp’ project. The zynq_fsbl_bsp project contains basic device drivers and standalone board support packages for the design created in Vivado. You can refer basic hardware drviers from this project. The Xiliinx UART device driver used in SLOS comes from this project. The zynq_fsbl_bsp project is a good reference for basic Zynq chipset configuration, but we don’t need to edit this project.

Open the zynq_fsbl project and go to *zynq_fsbl/src/* directory and open fsbl_hooks.c file. Then, search FsblHookBeforeHandoff() function. This function is called just before the FSBL bootloader jumps to the entry point of SLOS kernel.elf. You can’t see any meaningful implementations there. Modify that function as below.

```
#include "ff.h"
#include "sd.h"
#define TEMP_SCRATCH_ADDR 0x2000000
u32 FsblHookBeforeHandoff(void)
{
    char buffer[32];
    UINT br;
    u32 Status;
    u32 LengthBytes;
    u32 ImgCnt, i;
    FRESULT rc;
    FIL fp;
    char *boot_file;
    char *pScratch;
    Status = XST_SUCCESS;
    /*
     * User logic to be added here.
     * Errors to be stored in the status variable and returned
     */
    strcpy_rom(buffer, "ramdisk.img");
    boot_file = (char *)buffer;
    rc = f_open(&fp, boot_file, FA_READ);
    if (rc) {
            fsbl_printf(DEBUG_GENERAL,"SD: Unable to open file %s: %d\n",
    boot_file, rc);
```

```
        return XST_SUCCESS;
    }
    rc = f_lseek(&fp, 0x0);
    if (rc) {
        return XST_SUCCESS;
    }
    pScratch = (char *)TEMP_SCRATCH_ADDR;
    f_read(&fp, pScratch, 4, &br);
    ImgCnt = (u32)(*(u32 *)(pScratch));
    pScratch += 4;

    for (i = 0; i < ImgCnt; i++) {
        f_read(&fp, (void*)(pScratch), 4, &br);
        LengthBytes = (u32)(*(u32 *)(pScratch));
        pScratch += 4;
        rc = f_read(&fp, (void*)(pScratch), LengthBytes, &br);
        if (rc) {
            f_close(&fp);
            return XST_SUCCESS;
        }
        pScratch += LengthBytes;
    }

    if (rc) {
        fsbl_printf(DEBUG_GENERAL,"*** ERROR: f_read returned %d\r\n", rc);
    }
    f_close(&fp);
    fsbl_printf(DEBUG_INFO,"In FsblHookBeforeHandoff function \r\n");

    return (Status);
}
```

This function reads the content of ramdisk.img file from SD card and loads it to the *TEMP_SCRATCH_ADDR*. The TEMP_SCRATCH_ADDR is defined as 0x0200_0000. This function is also using the zynq_fsbl_bsp project's library functions such as f_open(), f_read() and so on. These are Xilinx's implementations in order to access the file system of SD card. We are not interested in those implementations. We already implement the SLFS working in ramdisk in previous chpaters. We want to load the ramdisk.img file to a specific location. The FsblHookBeforeHandoff() function is the place to be added to read and load the ramdisk image file. This function reads the ramdisk.img file based on the layout of figure 6-15. Line 36 ~ 46 considers the image total count and image size and loads the user applications's binary code into the proper memory address. You can still skip this implementation and use the downloaded zynq_fsbl_hook.elf for loading the ramdisk.img. The package command for BOOT.BIN now looks like below command.

petalinux-package --boot --fpga design_1_wrapper.bit --fsbl zynq_fsbl_hook.elf --u-boot=kernel.elf --force

6.5.4 Creation Application Files in SLFS

Now, we have SLFS, ramdisk image (ramdisk.img) and a ramdisk image loader (zynq_fsbl_hook.elf). After zynq_fsbl_hook.elf bootloader runs, the ramdisk contents are loaded into a TEMP_SCRATCH_ADDR and user applicatioins are ready to run. To run the user applications, we first have to create files of user application from the contents stored in TEMP_SCRATCH_ADDR. At the end of the start_kernel(), the create_ramdisk_fs() function is executed to access user application executables in the TEMP_SCRATCH_ADDR. create_ramdisk_fs() function creates application files in the SLFS. This ramdisk image was already loaded by the first stage bootloader into TEMP_SCRATCH_ADDR in the previous chapter. Creation of user application files is accomplished simply by using the file write of the contents of each application loaded in TEMP_SCRATCH_ADDR region. Since the contents of ramdisk image loaded at TEMP_SCRATCH_ADDR is same as the figure 6-15, we can parse it and write the binary data into a file of SLFS. The create_ramdisk_fs() function looks like below. This function is defined in *kernel/core/loader.c.*

```
int32_t create_ramdisk_fs(void)
{
/* variable declarations here */

        offset = 0;
        appCnt = *((uint32_t *)SCRATCH_BASE);
        offset += 4;

        for (i = 0; i < appCnt; i++) {
            szApp = *((uint32_t *)(SCRATCH_BASE + offset));
            offset += 4;
            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  }
```

The SCRATCH_BASE is the address of TEMP_SCRATCH_ADDR. Line 6 reads the first 4bytes which is the total application count stored in the ramdisk. Line 9 ~23 loops until the application count and writes the application binary data to a file of SLFS. In line 13, when application file is created, the name is chosen as ’App_index’. This name is used to access that file later. Line 14 opens a file in SLFS. Since this is the first time to access, open_file() function will create a new file and update all meta data in SLFS. Line 16 writes the binary content of user application into SLFS and close it in line 17. After this, the user application executable named *’App_index’* is placed at the ramdisk storage as a file of SLFS. The helloworld application in chapter 6.5.1 is named as ’App_0’. If there are more user applications in the ramdisk image, the SLFS executable files named *’App_1’, ’App_2’,* ... are placed in the ramdisk.

6.6 *ELF* Loader for User Application

One of the major operating system roles is loading applications and combining them with system hardware resources. In this chapter, we are going to load the helloworld user application and run it as a task like other kernel tasks. Every executable binary in SLOS is built as an *ELF (Executable and Linking Format).* We will develope a simple ELF loader to run a helloworld application.

6.6.1 *Elf (Executable and Linking Format)*

The *Executable and Linking Format* was originally developed and published by *UNIX*

System Laboratories (USL) as part of the *Application Binary Interface (ABI)*. The *Tool Interface Standards* committee *(TIS)* has selected the evolving ELF standard as a portable object file format that works on 32-bit Intel Architecture environments for a variety of operating systems. The ELF standard is intended to streamline software development by providing developers with a set of binary interface definitions that extend across multiple operating environments. This should reduce the number of different interface implementations, thereby reducing the need for recoding and recompiling code [3].

ELF is a common standard file format for executable files, object code, shared libraries and core dumps. By design, the ELF format is flexible, extensible, and cross-platform. For instance, it supports different endiannesses and address sizes so it does not exclude any particular central processing unit (CPU) or instruction set architecture. This has allowed it to be adopted by many different operating systems on many different hardware platforms [4].

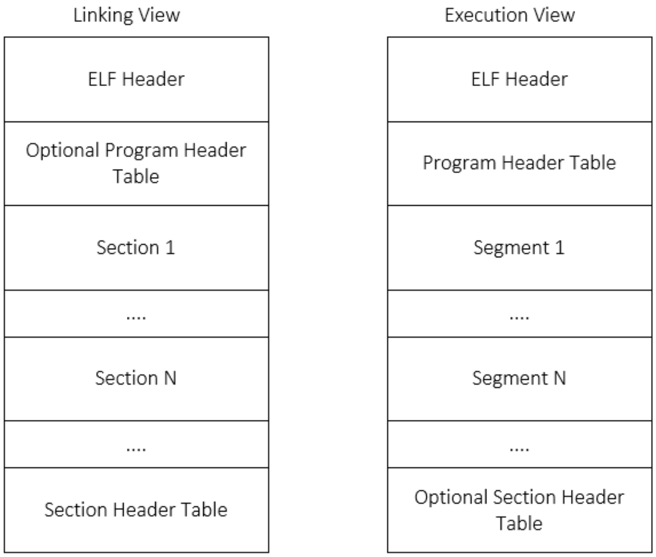


Figure 6-17: Object file format

ELF is a container file of binary code, data, string, symbols, relocation address and everything related to a linkable file or an executable file. Then, what are the linkable file and the executable file? When a compiler compiles a source code, it creates object files. Object files participate in either program linking (building a final program) or program execution (running a program). For convenience and efficiency, the object file format provides parallel views of a file's contents, reflecting the differing needs of these activities. Figure 6-17 shows an object file's organization [3].

As described in reference [3], an ELF header resides at the beginning and holds a *'road map'* describing the file's organization. Sections hold the bulk of object file information for the linking view: instructions, data, symbol table, relocation information, and so on.

A program header table, if present, tells the system how to create a process image. Files used to build a process image (execute a program) must have a program header table; relocatable files do not need one. A section header table contains information describing the file's sections. Every section has an entry in the table; each entry gives information such as the section name, the section size, etc. Files used during linking must have a section header table; other object files may or may not have one [3].

1) *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.

2) *Segment*

   Segment holds the information how to create a runnable process image. Multiple sections can be integrated into one segment. A *Program Header Table* has the entries of segments.

ELF header is the first place to describe these contents. An example of runnable ELF file layout looks like figure 6-18. This example ELF has also an optional section information.

ELF header has the top-level information including the Section Header Table and Program Header Table. It has the ELF Identificatioin, object file type, required architecture machine type, the entry point, program header table location and size, section header table location and size, and so on.

Each section is referenced by Section Header (section name, type, address, offset, sh_link, size etc.) and Section Header Table. Section header table entry is composed of section name, section type, flags, address, offset, size, sh_link, sh_info, alignment, and entry size. To access a specific section, we need to go to ELF header -> section header table -> section.

The ELF of a runnable program should have program header. Program header describes the segments which has the information about loading into the memory such as segment type, file offset, virtual address, physical address, and so on. We can access all program segment by going through ELF header->Program header->Segment.

One thing we need to keep in mind about segment for a runnable program is the *Base Address.* Base address is the lowest address of the segment when it is loaded into memory. Since our user application is built based on address 0x0, the virtual addresse used in the application's segment can be considered as an offset relative to address 0x0. When we load a user application ELF into memory, we need to take care of this. We will cover this in next chapter.

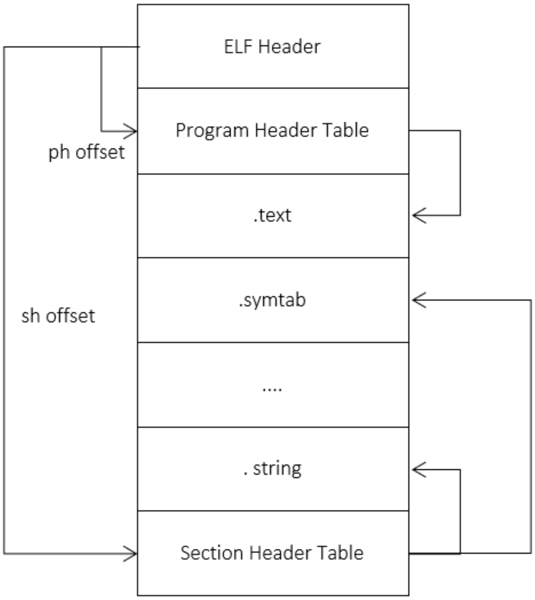


Figure 6-18: An example of ELF file organization

6.6.2 Loading Helloworld Application ELF

We have created an application executable and a library for system call, then put them together in ramdisk. We also have loaded the ramdisk into SCRATCH_BASE region and created file of SLFS in chapter 6.5. In this chapter, we will load it into memory and run it. The functions in *kernel/core/loader.c* are for loading the user application. load_ramdisk_app() reads the application file from SLFS, interprets the ELF headers, finds the entry point of the application and runs a forkyi() to launch the application. load_ramdisk_app() function is:

```
void load_ramdisk_app(uint32_t appIdx)
{
    /*
     * do basic stuffs
     */

    app_load_addr = (char *)(USER_APP_BASE +
                    APP_LOAD_OFFSET +
                    USER_APP_GAP * appIdx);

    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 memory address for application loading region which is 0x0180_0000. Line 12 finds the application file named ’App_0’ which is our ’helloworld’ application created in chapter 6.5.1. If finding the application file is successful, line 14 reads the byte code of helloworld application from SLFS and copies it to the loading address 0x0180_0000. Next, the line 15 parses this ELF file and runs the executable. The steps of running the user executable is finding the application’s main entry point and calling the forkyi() function with that entry point. To find the location of main() function in helloworld, load_elf() uses the string table and symbol table in the ELF file. Below is the load_elf() function implementation.

```
task_entry load_elf(char *elf_start, uint32_t idx)
{
/*
 * variable declaration and initialization here
 */

    hdr = (Elf32_Ehdr *) elf_start;

    exec = (char *)(USER_APP_BASE + idx * USER_APP_GAP);
    phdr = (Elf32_Phdr *)(elf_start + hdr->e_phoff);

    for (i = 0; i < hdr->e_phnum; ++i) {
        if (phdr[i].p_type != PT_LOAD) {
            continue;
        }
        if (phdr[i].p_filesz > phdr[i].p_memsz) {
            xil_printf("load_elf:: p_filesz > p_memsz\n");
            return 0;
        }
        if (!phdr[i].p_filesz) {
            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.

1) First part is finding the loadable segment in the program headers and copies that segment into the execution memory region.

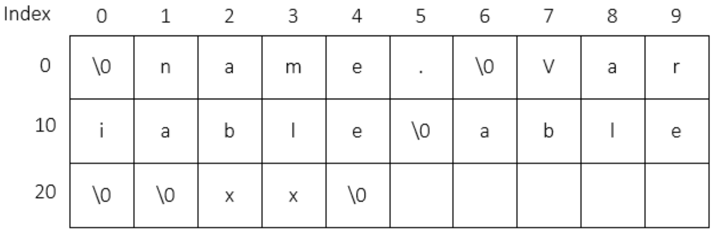
2) Second part is the part that figure out the main entry address by browing the *'main'* string in the symbol table.

Line 12 ~ 28 copies the loadable segment to the memory. This routine goes through all the program header entries. While it traversing the program header table, if the type of the segment (p_type) is PT_LOAD, then it copies that segment to the execution memory which starts at 0x0100_0000. When copying the PT_LOAD segment to the execution memory region, the segment virtual address (p_vaddr value of the segment) must be added to the base address. The p_vaddr value is a relative value from the base address. The base address in this case is 0x0100_0000. The line 24 ~ 27 is for copying the PT_LOAD segment considering the virtual address offset from base address.

Now, we have loaded the execution segment into the memory. Next step is finding the entry point of the runnable segment. Finding the entry point is locating the main function. This can be done through the symbol table in section header. Line 31 ~ 38 is the routine finding the address of main entry point. For this, it first finds the symbol table section by traversing the section header table. The section header type (sh_type) of symbol table is defined as *SHT_SYMTAB* which is defined as 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 index in the section header table for the 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 section holds 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) in section header and the symbol name field (st_name) in symbol table entry hold an index of this string table.

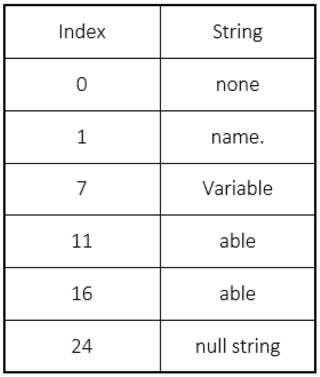
Following figure shows an example of a string table with 25 bytes and the strings associated with various indexes.



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					

Figure 6-19: Example of string table containing string "*name.", "Variable", "able", null, "xx"*

An example of corresponding index for string table entries is depicted in figure 6-20. As the figure 6-20 shows, a string table index may refer to any bytes in the section. A string may appear more than once; references to substrings may exist; and a single string may be referenced multiple times. Unreferenced strings also are allowed.



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

Figure 6-20: String index example; An arbitrary index value 16 points string *"able"*

By using this string table and symbol table, the find_sym() function in line 34 tries to 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 it succeeds in finding the main symbol's entry in the symbol table, it returns that symbol's st_value which is the virtual address of the main symbol.

After finding the entry point address, the create_usr_cfs_task() function in the load_elf() function forks a new task with the main entry. This just calls a forkyi*()* and creates another CFS task. Now, the helloworld application is scheduled by the CFS scheduler with other CFS tasks.

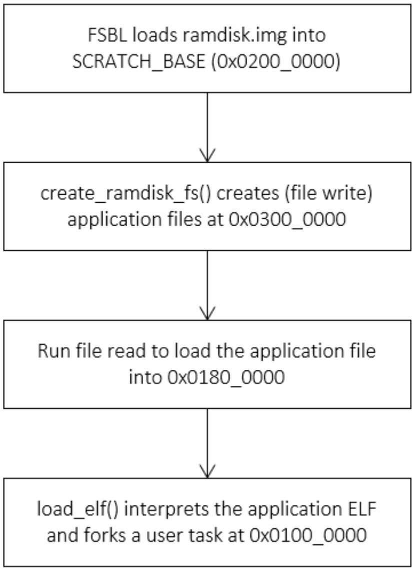

FSBL loads ramdisk.img into SCRATCH_BASE (0x0200_0000)
create_ramdisk_fs() creates (file write) application files at 0x0300_0000
Run file read to load the application file into 0x0180_0000
load_elf() interprets the application ELF and forks a user task at 0x0100_0000

Figure 6-21: Helloworld user application load sequence

Figure 6-21 summarizes the flow from loading ramdisk image file into CFS user application task. The flow starts from FSBL copying ramdisk contents into 0x0200_0000 and the final user task is relocated at 0x0100_0000 with a 1MB gap between user tasks. The relocated user application has relative address offset from its base address.

6.7 Putting it altogether

To see the message from the helloworld user application, we first changed the build sequence. The build system changes occur in Makefile*.* Then the implementations from chapter 6.2 to chapter 6.6 launches the user application. The Helloworld application prints simple messages into serial terminal. That’s all. Let’s load a Helloworld application and print a message through the system call.

6.7.1 Changes in The Build Process

The build process in figure 3-6 has some changes for building ramdisk.img. The build process now builds two target images which are ramdisk.img and kernel.elf. First, it builds the libslos.a library and mkfs binary. libslos.a is a library for system call. This library is statically linked into Helloworld application. SLOS doesn't have a dynamic library loader. This library has two source files; print_mesg.c and syscall.S in *libslos/* directory.

mkfs executable is used for building the ramdisk.img. This executable is a Linux application and built g++ compiler, not using cross-compiling. Except the mkfs binary, all other binaries are built by ARM cross-compiling toolchain. The mkfs concatenates the applications and build the ramdisk in figure 6-15.

The second part building kernel.elf and BOOT.BIN stays same. When building BOOT.BIN, we must use a new zynq_fsbl_hook.elf. There are new sources for storage management which is automatically covered by Makefile.

Now, we need to copy 2 images (BOOT.BIN, ramdisk.img) to SD card. ramdisk.img file has the helloworld application. We also need to use the new prebuilt fsbl (or you can build it by yourself by following steps in chapter 6.4.3) which loads the ramdisk image to 0x0200_0000. For these changes in the build process, there is a few modifications in the Makefile. Refer a Makefile source for detailed changes. The whole build process for storage management looks like figure 6-22.

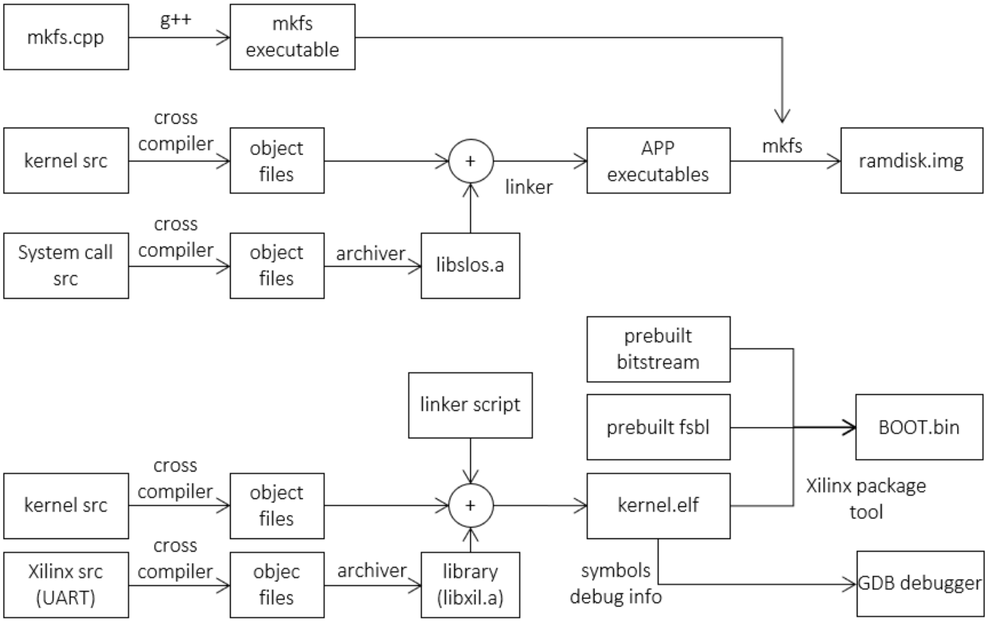
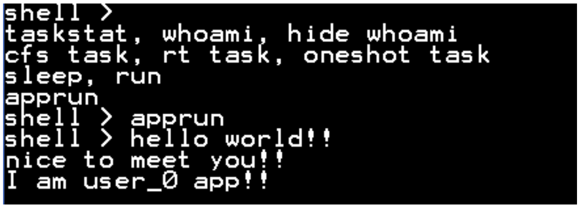


Figure 6-22: A new build process for ramdisk.img and libslos.a library

6.7.2 Let's see it

In the SLOS project repository, run following command:
git checkout SLOS_CH6

Then, run make command in the slos directory. This command builds every binaries including mkfs, libslos.a, ramdisk.img and kernel.elf. All binaries will be created in a corresponding directory under *out/*. Build the BOOT.bin with prebuilt FPGA bitstream and new FSBL. The new FSBL can be downloaded from tools repository. Copy the *out/ramdisk/ramdisk.img* and BOOT.bin into SD card. 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 as below figure 6-23.



```
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!!
```

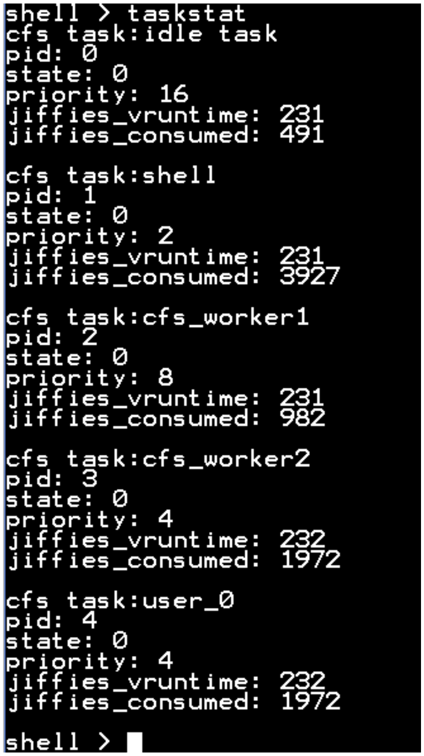
Figure 6-23: Helloworld messages from *apprun* command

The helloworld application prints messages through a write system call we implemented to access system hardware resources. If you see those message in the terminal, it means the system call is working in SLOS.

Now, run the *'taskstat'* command to see the CFS task status. User application is run as a CFS task and it should be served fairly in virtual runtime perspective. As you can see in figure 6-24, the virtual runtime is pretty similar among CFS tasks which means those tasks including user application are sharing the cpu time in quite fair way.

6.8 Summary

In the first half of this chapter, we discussed the SLFS (Simple and Light File System). SLFS is composed of super block, inode bitmap, data block bitmap, inode table and data block. SLFS's super block doesn't have meaningful information but it can be extended someday in the future. SLFS implements the inode to represent a file. Inode is a meta data storing all information needed to maintain a file. Bitmaps for inode table and data block are supported to keep track of the allocation of blocks. Since the size of each block of SLFS 512 bytes, the file size is limited. To overcome this file size limitation, SLFS supports 2-level data block. With



```
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 > 
```

Figure 6-24: CFS task status after running user application

this, SLFS supports the file size up to about 8Mbytes which is much bigger than the max ramdisk size. SLFS supports file open, close, read and write. These file operations are used in mounting the contents of ramdisk into init file system and loading the executable ELF.

The second part is for launching the user application. For this, we created a ramdisk.img and uses a SLFS file system. To create the ramdisk.img, we first have to edit the Makefile to change the build system and tweak the Xilinx FSBL bootloader. In order to launch user application, a very simple ELF loader was implemented. ELF loader loads user application by using file system's read() api and loads the user application to a specific memory location. Along with the ELF loader, a libslos.a library for system call was implmented for user application and their functionality was demonstrated.

Reference

[1] https://en.wikipedia.org/wiki/Inode

[2] https://www.kernel.org/doc/Documentation/filesystems/vfs.txt

[3] Portable Formats Specification, version 1.1, Tool Interface Standards (TIS)

[4] https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

[5] ARM Cortex-A Series Programmer’s Guide

7 Hardware and Software Codesign

7.1 Introduction

As I mentioned earlier, the main job of operating system is driving the system hardware and exposing those system resources into user applications. The system hardware resources include all hardware installed in the system such as CPU, memory, storage, I/O peripherals, network and so on. So far, we assume a generic hardware; ARM-Cortex A9 processor for process management, MMU for memory management, UART device driver, ramdisk. These hardwares are already defined by the manufacturers for general usages.

But most embedded systems have its own specific requirements. For example, mobile phone needs network and highly optimized power consumption. Embedded system for the automobile must have a low latency and stable operations. Some embedded system for the factory automation needs realtime response to handle the information from different sensors.

A general-purpose processor or hardware might not be good enough to satisfy these specific needs but rather be good for diverse applications from different industries. For the custome applications, we can see better achievement (performance, power consumption, latency etc.) by designing special hardware or coprocessor to hit the goal.

When we design this custom system to meet a specific requirement, we first have to partition the behavioral requriements between software and hardware. If hardware implementation would show better fit, then put that part's implementation into the hardware and design the custom, specific hardware. If software implementation is more appealing, then develop a software application and put that applicatioin into a general application processor. While doing this, we also have to define the interface between them for the cooperation of hardware and software. Moreover, the requirements of these systems sometimes change after shipping the product to the customer. That means the shipped hardware needs to be reconfigurable to meet those requirement changes.

With these being said, Xilinx Zynq products really fit well into these hardware-software codesign requirements. As we talked in chapter 2.7, Zynq chipset has a PS subsystem for a general-purpose programming and also has a PL subsystem which is for programmable and reconfigurable hardware development. Xilnx Zynq uses AXI bus to interface PS subsystem and PL subsystem. This means the AXI bus interfacde and its communication protocol are used at the bottom layer for our hardware and software interface. For the detailed information of AXI protocol, refer the document [2], and [3]. On top of AXI interface, we can program register to configure hardware operation and hardware can trigger interrupt to inform the running software of hardware event. The Cortex-A9 ARM processor for PS subsystem itself is already quite good for the most of generic requirements. In addition, we still can design our own hardware for a better fit for some special needs. For the example of hardware-software codesign in Zynq chipset, we can use ARM processor and Linux for receiving data through TCP/IP connection from server computer and the FPGA code in PL system can control peripheral hardware with the received data. We can develop a device driver or an application to interface the PS and PL system.

In this chapter, we will experience two simple examples on hardware-software codesign. We are going to partition the requirements, design our own hardware peripheral in PL subsystem and implments the interface between our special hardware and software; programming registers in peripheral, interrupt signal from peripheral to device driver. We will add new device drivers. These device drivers are a software part of this hardware-software codesign.

After chpater 7, we can build the whole, complete subsystem blocks like in figure 7-1. Newly added blocks are marked with grayed rectangle. Those are *DMA device driver, outstream device driver, Modcore processor and outstream device.* DMA device driver and outstream device driver cover the software part and the Modcore processor, outstream device perform the hardware part. The Modcore processor and outstream device are implemented by using a *VHDL (VHSIC Hardware Description Language)*. This book doesn't cover the VHDL language. Refer many other good books [1] or articles for this. In order to add the VHDL implementations for Modcore, follow steps in chapter 7.2.4. But the SLOS has *TCL* scripts to generate the Vivado project automatically for the Modcore and Outstream device. You can easily use them to create the VHDL project in Vivado 2018.3 (The latest master branch has 2019.2). Vivado version upgrading in the script can be done also.

Let's check out the SLOS for the chapter 7.

```
git checkout SLOS_CH7
```

New source files are added as below.

1) *genesis*

   This directory stores all files for Vivado project, VHDL source files for Modcore and outstream.

2) *genesis/ip_repo*

   This directory is for VHDL hardware *IP (Intellectual Property)*. Currently, it has Modcore and outstream device.

3) *genesis/ip_repo/modcore_1.0/hdl/modcore_v1_0.vhd*
   This is a VHDL file that has a top-level hdl wrapper implementation. This file has instances of master AXI interface and slave AXI interface.

4) *genesis/ip_repo//modcore_1.0/hdl/modcore_v1_0_M00_AXI.vhd*
   This is a VHDL file that has a master AXI interface implementation. The implementation includes a custom DMA and a simple 256's complement operation for demonstration.

5) *genesis/ip_repo/modcore_1.0/hdl/modcore_v1.0_S00_AXI.vhd*
   This is a VHDL file that has a slave AXI interface implementation. The master of this is PS subsystem's AXI master. The Modcore device driver implements the PS subsystem's AXI master logic. This slave AXI interface has some registers to control the Modcore hardware.

6) *genesis/ip_repo/odev_1.0/hdl/DataConsumer.vhd*
   This is for the simulation of data consuming in outstream device. The outstream data can go out to other periphereal hardware through external pin connection. The DataConsumer module plays the role of external hardware.

7) *genesis/ip_repo/odev_1.0/hdl/Itab.vhd*
   This module is a information table to store the metadata of a data block. This data block and metadata are normally generated by an application running in PS. Since this is a temporal storage, it needs a flow control between ARM AXI slave and AXI master module.

8) *genesis/ip_repo/odev_1.0/hdl/RdBuff.vhd*
   This module buffers the data stream from memory. This module also needs flow control between AXI master and DataConsumer.

9) *genesis/ip_repo/odev_1.0/hdl/odev_v1_0.vhd*
   This is the top view of outstream device. It consists of submodules and their connections.

10) *genesis/ip_repo/odev_1.0/hdl/odev_v1_0_M00_AXI.vhd*
    This is a master AXI module to initiate the read data from memory. It gets the metadata from the information table and saves the data to read buffer.

11) *genesis/ip_repo/odev_1.0/hdl/odev_v1_0_S00_AXI.vhd*

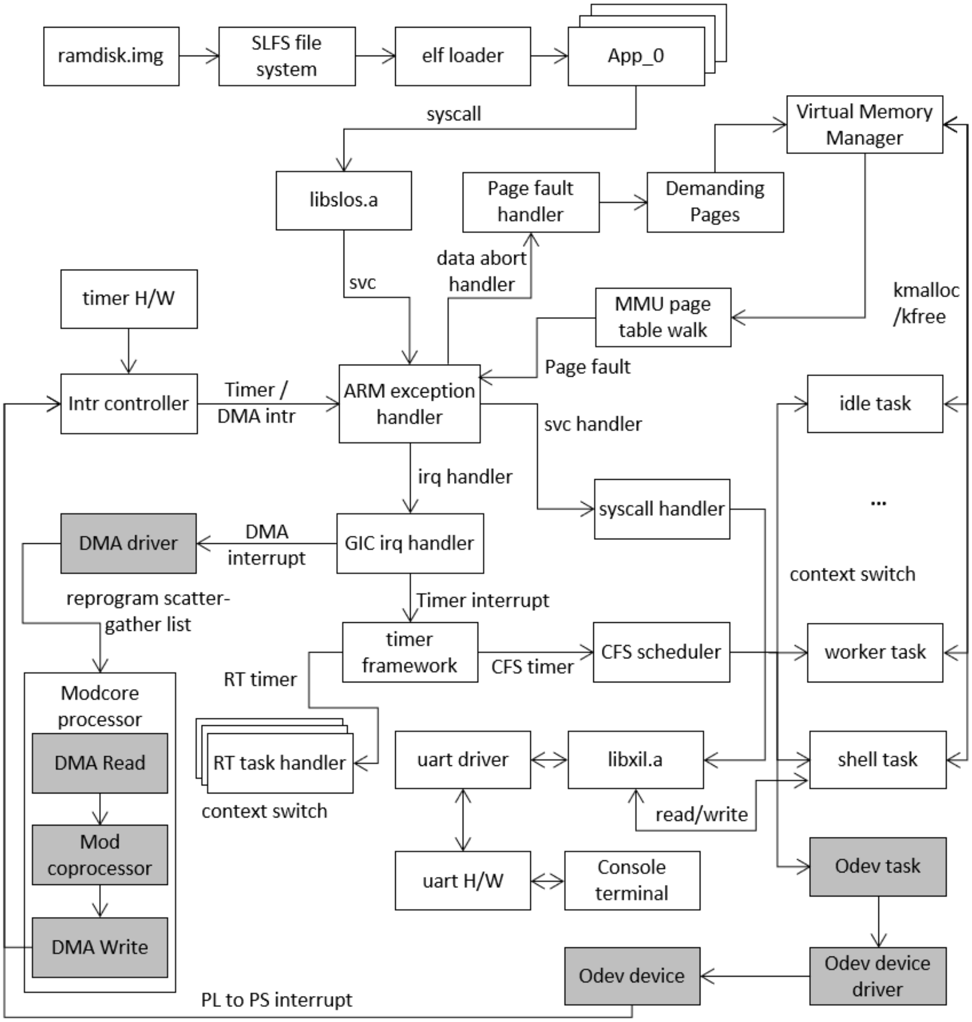


Figure 7-1: Top view after adding custom hardware and its device driver

This is a slave AXI module saving the metadata to information table sent from application running in PS system. It has 8 programmable registers exposed to an application.

12) *kernel/drivers/dma/dma.c*
This is a device driver for the DMA hardware. It initializes the Modcore hardware and handles the *dma_irq* interrupt.

13) *kernel/inc/dma.h*

Header file for dma device driver. This file defines the address of memory-mapped registers of Modcore hardware.

14) *kernel/drivers/drivers/odev.c*
This file has the implementations for outstream device driver. It has initialization, start, stop out stream, adding Itab entry and outstream interrupt handler.

15) *kernel/inc/odev.h*
This file has AXI slave register bit definitions and out stream device driver's function declarations.

In addition to these new files, two new interrupts (*dma_irq, odev_irq)* from PL subsystem are registered to the GIC interrupt controller.

We will practicea simple designfor hardware and software codesign. This hardware and software will perform following two operations;

1) Read data from source memory region and process some operation (256's complement) on every byte data and write back the result into destination memory area. This works like a *fire-and-forget* until there is a done interrupt for PS system.

2) Keep adding the information table (Itab) entry with source memory address and length and have the AXI master read and transfer data to DataConsumer module. Software keeps program next work.

The Cortex-A9 ARM processor in PS subsystem also can do the first job by running an application. But even this simple operation, the byte-wise operation of memory region is very expensive; it takes long to complete. Moreover, using PS subsystem means this operation should share the CPU occupation time with other tasks such as shell task, worker task and so on. This results in worse performance and lowers the possibility to complete the job within the designed deadline. This example, transferring the data to the next module after simple bit operations within the deadline, could be common in some I/O bounded embedded system and this module is a coprocessor in running a calculation to lower the load of main processor.

The second practice is also directly access to the memory region and transfer the data to external peripheral device. But this example just uses a DataConsumer for the external consumer device. This happens continuously and makes out stream to the consumer by having a dedicated task add an entry to information table and AXI master keep pulling the data to consumer device. There is a flow control to manage the overflow and underflow in each buffer (Itab, RdBuff).

7.2 Design a Modcore Coprocessor in PL Subsystem

First, let's look into the Modcore coprocessor. To meet this type of design, we have to partition the implementation into hardware portion and software portion. We move the time critical operations into hardware implementation and the software just program the hardware and delegates the logical operationto the hardware. The hardware implementation is accomplished by designing the Modcore coprocessor in PL subsystem. Once it is programmed, the Modcore coprocessor doesn't affect the PS subsystem's work load. No CFS scheduling, no CPU time sharing with other worker tasks; it works on its own.

The Modcore coprocessor is composed of 3 subsystems; *Modcore Topview module*, *AXI Master module* and *AXI Slave module*. AXI Master module has a DMA module and runs a simple modulo logic operation. The DMA module reads memory data from source address and stores it into its local temporary storage, calculates 256's complement of every byte and store the results back into the memory of destination address.

The application or task in the ARM processor in PS subsystem needs to program this coprocessor and handles the interrupt occurring whenever the programmed transaction is done. Programming the DMA is done through the registers in AXI slave module. DMA device driver running in ARM processor programs this Modcore processor by using the registers in AXI slave module.

The genesis/ip_repo*/modcore_1.0/hdl* directory has the VHDL sources for Modcore processor. Modcore source files are based on the Xilinx example source code of AXI Master and AXI slave module. This book doesn't describe the changes in the original AXI master and slave sources but with a small number of changes, we can develop a simple DMA module and add a custom logic into it.

Most of the logic including the DMA logic are implemented in the AXI master module. AXI master has its state machine and initiate the AXI transaction actively. On the contrary, the AXI slave mostly work with device driver module in PS subsystem in a passive way. It gets work order from DMA device drvier and saves it to its registers; allows the device driver to set the control register, to read status register. A high-level block diagram among DMA device driver in PS subsystem and Modcore modules in PL subsystem, and external memory is described in below figure 7-2.

As in figure 7-2, the DMA device driver works with the registers in AXI slave. AXI slave owns the Modcore registers and passes the register information into AXI master. AXI master works with AXI slave and RAM memory both. The communication among hardware subsystems (Modcore, RAM, and PS subsystem) is achieved through the AXI bus interface. In figure 7-2, we define the software and hardware codesign in high level. The interface from hardware (Modcore processor) to software (DMA device driver) is achieved through the interrupt. The opposite interface from software to hardware can be done through the programming of registers in AXI slave.

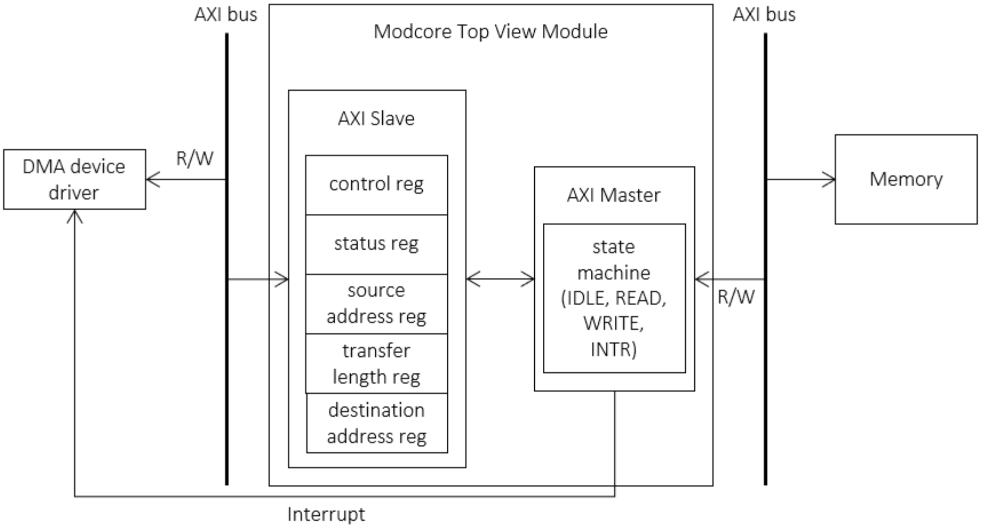


Figure 7-2: Top view for DMA device driver, AXI Slave, AXI Master, AXI bus and Memory

Now, we define hardware, software and the interface between them. Then, let's first implement the hardware part of this hardware-software codesign practice.

7.2.1 AXI Master's DMA Module

AXI master block initiates the data transaction between Modcore processor and the RAM memory. It reads data from memory, saves it into its block memory, performs some meaningful operations, and writes back the result into the memory. Once the source and destination memory regions are set by a task in PS system, this memory R/W operation is performed without any intervention of ARM processors in PS.

Figure 7-3 depicts the operations of AXI master block. AXI master block works with RAM memory controller for data access and generates an interrupt to the PS subsystem when all its jobs are done. AXI master core is composed of *DMA Read* block*, DMA Write* block and *MOD operator* block. The *MOD operator* block performs the 256's complement operation.

Since AXI master has to initiate the work with memory controller, it has a state machine to handle each phase of the transaction.

The AXI master has four state and its state machine looks like figure 7-4.

1) *IDLE* State

   AXI master initializes all its local variables and wait the triggering signal from software (DMA device driver) in the PS subsystem.

2) *READ* State

   AXI master starts to read data from external RAM memory. AXI master has a *burst*

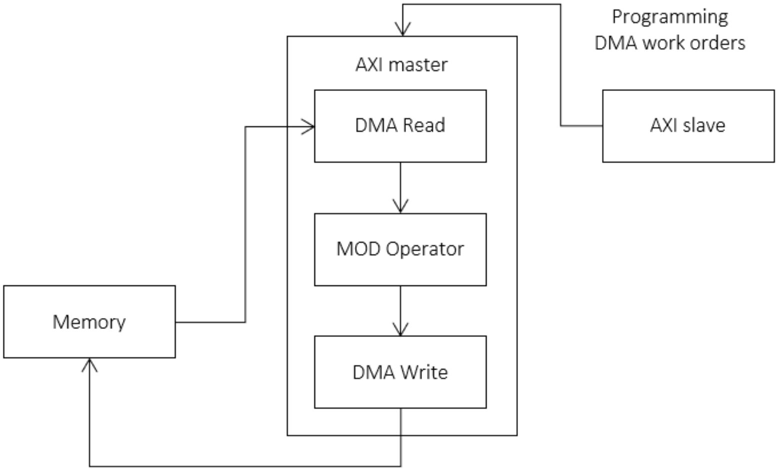


Figure 7-3: Operation flow in AXI master block

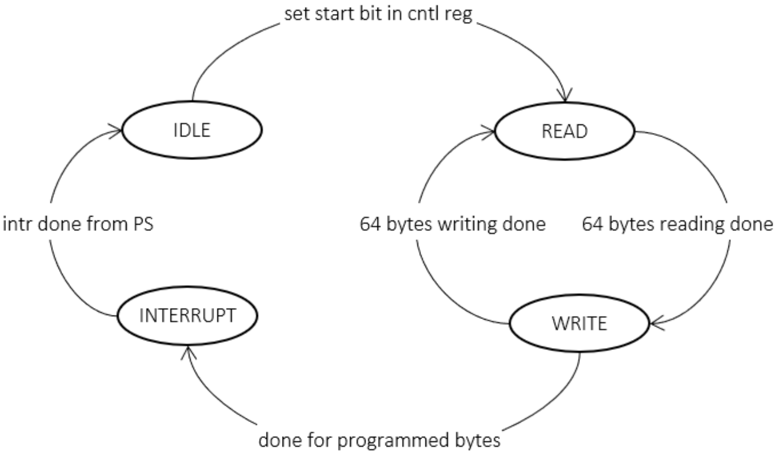


Figure 7-4: State machine of AXI master module

mode for the transaction with AXI slave. In this case, the memory controller works as an AXI slave. There are 1, 2, 4, 8, 16, 32, 64, 128, and 256 burst lengths and each burst is composed of multiple *beats* and each beat is 32bits (4bytes) wide. Modcore processor is using 16 burst mode and one burst transaction can read 64 bytes (burst length (16) times beat size (4 byte)) at a time. AXI master saves the read data to the *register file* which is same size of 1 burst transaction.

3) *WRITE* State

AXI master goes to WRITE state after one burst transaction is done. Just before writing back to the destination address in memory, the AXI master runs a simple modulo operation for each byte as below.

byte[i] = 256 – byte[i] % 256;

This simple VHDL code does 256's complement for each byte. In this Modcore example hardware, the test logic is very simple, but adding more complex logic in practical application can be done in a similar way. After finishing write back to the destination memory, WRITE state checks the amount of total data. If there are still more data to handle, it goes back to *READ* state. But if there is not more data to read, the WRITE state goes to *INTERRUPT* state.

4) *INTERRUPT* State

There are two interrupt lines which are physically connected from Modcore to GIC controller of PS subsystem. One for Modecore processor and the other for out stream device. Tnterrupt number from PL to PS in Zynq chipset was mentioned in chapter 4.4.5 briefly. Modcore coprocessor uses an interrupt #61 with *level triggering*. When Modcore finishes all its job, it raises this interrupt line to high and wait *interrupt_done* signal from the PS subsystem. The interrupt handler in DMA device driver sets *interrupt_done* bit in control register of AXI Slave. This makes the Modcore lowers the interrupt line and goes into the IDLE state.

The DMA module needs only source address, destination address and length for its operation. Once the application in PS subsystem programs this information into the registers in AXI slave, the DMA module starts to work by itself with its state machine. After issuing the DMA work order, the application can return to its original work and proceed without blocking by the operation. ARM processor in PS subsystem and Modcore in PL subsystem runs independently. The application is not blocked by this Modcore operation and keep going after configuring the Modcore. Since Modcore generates an interrupt when it finishes its job, DMA device driver knows the right time when it can program next Modcore's work or when the application processes the results of work done. The application keeps running after delegating its heavy job to Modcore processor and handles the job done event when Modcore processor completes the job. This is the basic idea of hardware-software codesign.

7.2.2 AXI Master's Modulo Operation Module

Modulo operation module calculates the 256's complement of each byte in memory. For example, if the value of a byte is 1, then the result of this operation is 255, if the value is 2, the result is 254 and so on. This is easily done by using VHDL. We can add more complex logic blocks and connect it to DMA module. The DMA module is also easily extended and custom logic can be added. In our simple OS, we just demonstrate a very simple logic operation.

7.2.3 AXI Slave Module

AXI slave module doesn't have a state machine. Instead, it receives the master's work orders coming from device driver in PS subsystem. Since AXI slave module is a slave block, the logic that AXI slave needs to perform is implemented in DMA device driver which is a master in this communication.

The AXI slave module in Modcore processor has five registers (*control, status, src_addr, len, dst_addr)* for the DMA device driver to program its logic. AXI slave registers are described in the table 7-1.

Register Name	Address (Offset)	Width (bits)	Description
control	0x00000000	32	Control register
status	0x00000004	32	Status register. Not used.
src_addr	0x00000008	32	Source address register
len	0x0000000c	32	Transaction length register
dst_addr	0x00000010	32	Destination address register

Table 7-1: Modcore AXI slave register description

The address of these registers in the table 7-1 is offset according to the base address of Modcore processor. The base address is preset by the Vivado's *Address Editor*. The bitstream for basic Zynq chipset in chapter 2.3.7 which is uploaded to the *tools* github has 0x43C1_0000 for the Modcore processor's base address.

The details of *control* register are described in table 7-2. Only bit[0] and bit[1] in *control* register are used in this practice. These bits are programmed by the device driver.

Field Name	Bits	Type	Description
reserved	31:3	RW	Reserved
RESET	2	W	Reset bit. This bit makes all modcore subsystems go through reset procedure.
INTR_DONE	1	RW	Interrupt done bit. DMA device driver in PS must set this bit when DMA interrupt service routine completes its job and returns. Once this bit is set, the AXI master state goes to IDLE state.

START_DMA	0	RW	DMA transfer start bit. DMA device driver sets this bit when it triggers the Modcore to run the DMA transfer and modulo operation.

Table 7-2: Modcore control register bit description

The *Status* register is *read-only* register and was intended to represent the status of Modcore processor and to be set by the Modcore Processor. But it is not used in this practice design. This feature can be added later.

src_addr and *dst_addr* registers are used to set the physical address of source memory region and destination memory region. The *len* register is the number of bytes from source address copied to destination address with modulo operation.

These registers in AXI slave module are set by DMA device driver except the *Status* register. AXI slave relays the register values to AXI master module from DMA device drvier. The signal connections between AXI slave and AXI master are implemented in the top module of the Modcore processor defined in *modcore_v1_0.vhd.*

7.2.4 Adding Modcore to Vivado and Building BOOT.BIN

You can easily add your custom hardware by tools->create and package a new *IP (Intellectual Property)* in Vivado IDE. You can also refer Xilinx user guide document *[4]* for a detailed description on adding a custom IP. If you don't have interests in building your own hardware, you can directly download the bitstream from *tools* git repository as below or you can skip this chapter (In next example of outstream device, we will automatically generate the Vivado project for Modcore and outstream device by using *TCL* script).

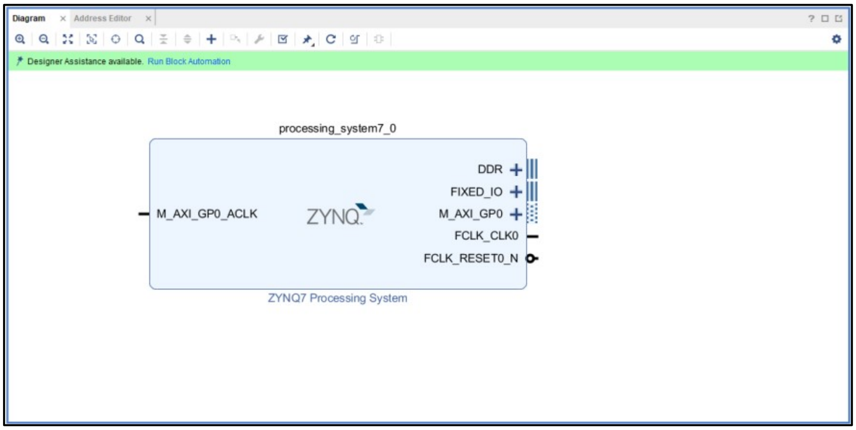
git clone https://github.com/chungae9ri/tools

This *tools* repository has a *design_1_wrapper.bit* bitstream for the PL subsystem. You can run a *petalinux-package* command to build a *BOOT.BIN* containing the Modcore hardware bitstream.

If you want to build the FPGA bitstream with Modcore coprocessor, follow below steps. This process step is based on Vivado 2018.1 version.

1) Run Vivado 2018.1.
2) Create a new project in Vivado File -> new -> Project. Enter project name modcore, select target language as “VHDL” while going next up until “Default part” selection menu.
3) In “Default part” selection window, choose “boards” -> “ZYNQ-7 ZC702 Evaluation Board”. If you are using different board or chipset, choose a proper one. This book assumes ZC702 evaluation board.
4) go next -> finish. Now, a bare, simple Vivado project was created.

5) In the “Flow Navigator” pane in the modcore project, click “Create Block Design” menu.
6) Press “+” button *(add IP* button*)* and type “Zynq” in the search box. Then Vivado will show “ZYNQ7 Processing System”. Double click it to add it into block design.
7) You should see a diagram window as below screen capture.



Designer Assistance available. Run Block Automation

Figure 7-5: Block design after adding *ZYNQ7 Processing System*

8) Click “Run Block Automation”. It will automatically connect external DDR and Fixed IO.
9) Connect FCLK_CLK0 clock to M_AXI_GP0_ACLK pin. You can easily do this by mouse clicking on FCLK_CLK0 pin and drag mouse to M_AXI_GP0_ACLK pin.
10) You should have a block design as below.

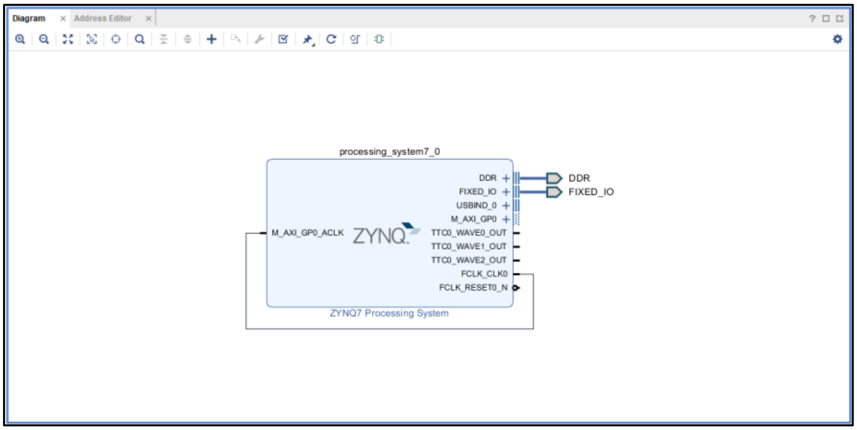
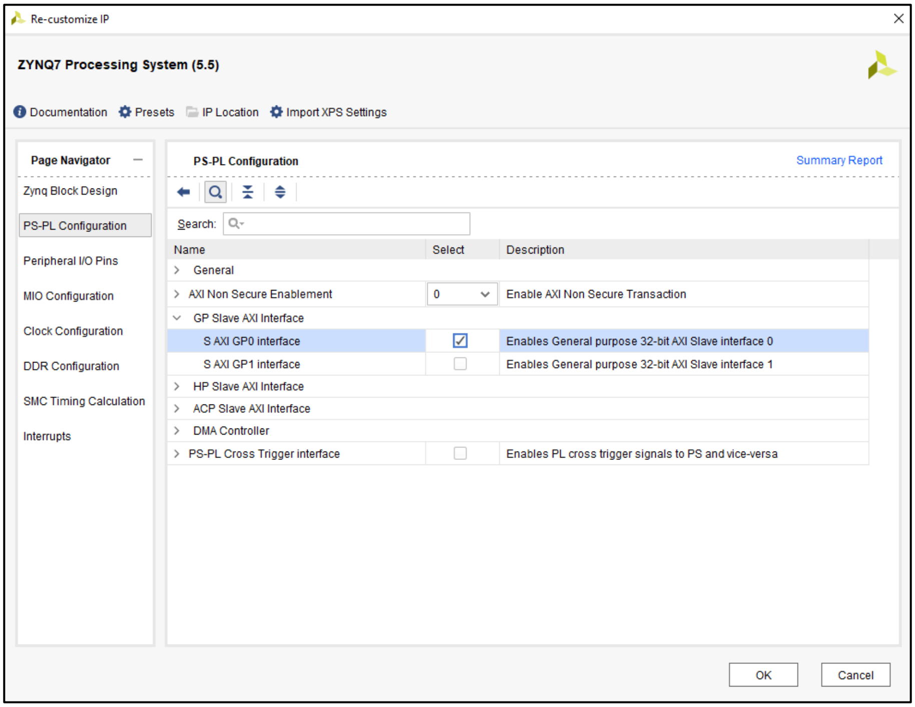


Figure 7-6: Block design after connecting the *FCLK_CLK0 with M_AXI_GP0_ACLK*

11) Now, let’s perform some customization on the ZYNQ processor. First, add an AXI slave

module. Double click the ZYNQ module in the Block Diagram. Now you can see the Re-customization popup window. Go to PS-PL Configuration -> GP Slave Interface and check “S AXI GP0 Interface”. Later, this slave interface will be connected to Modcore processor’s AXI master interface. It looks like figure 7-7.

12) Second, let’s add an PL interrupt to the ZYNQ processor. Later, we will connect the interrupt from Modcore hardware to this interrupt. Go to Interrupts and enable PL-PS interrupt as below figure 7-8. Then, click OK.



Name	Select	Description
› General		
› AXI Non Secure Enablement	0	Enable AXI Non Secure Transaction
⌄ GP Slave AXI Interface		
S AXI GP0 interface	☑	Enables General purpose 32-bit AXI Slave interface 0
S AXI GP1 interface	□	Enables General purpose 32-bit AXI Slave interface 1
› HP Slave AXI Interface		
› ACP Slave AXI Interface		
› DMA Controller		
› PS-PL Cross Trigger interface	□	Enables PL cross trigger signals to PS and vice-versa

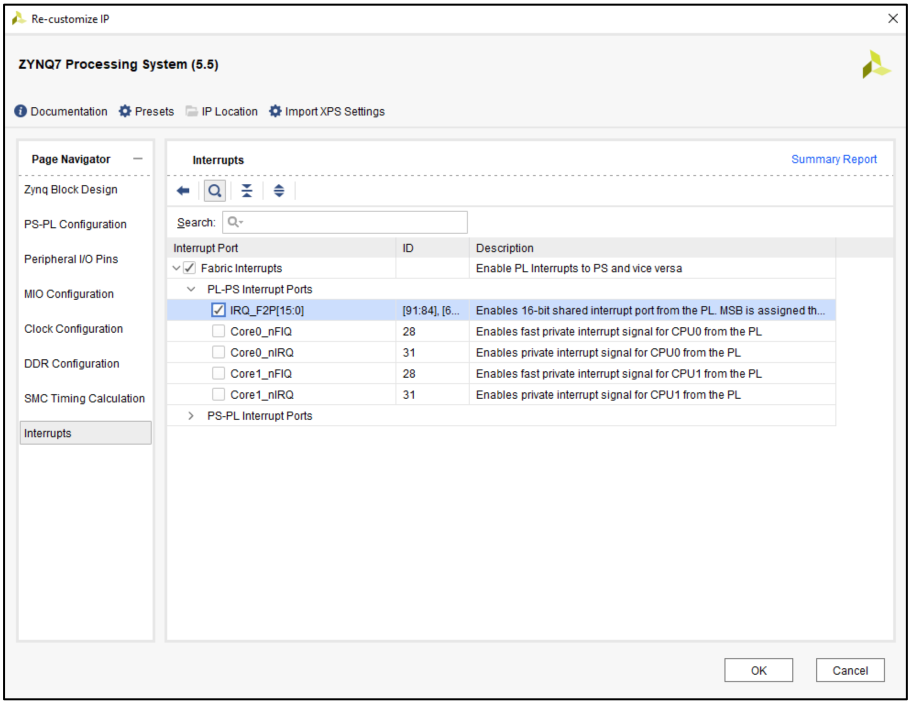
Figure 7-7: ZYNQ7 block customization – adding AXI slave to PS subsystem

13) Now, it’s time to add our custom hardware. Click Tools -> Create and Package New IP... Then, click next -> check “create a new AXI4 peripheral” under Create AXI4 Peripheral and click next -> input “modcore” in the name field. Click next -> In Add Interfaces window, click “+” button and edit the interface field as figure 7-9. Then, click next.

14) Check “Edit IP” and click Finish button. Vivado will open a modcore edit window. In the source pane, we can see the three vhdl sources in figure 7-10.

a) modcore_v1_0.vhd

The top view of modcore processor. It has one AXI slave and one AXI master instance and connects the port signals between them.



Re-customize IP

ZYNQ7 Processing System (5.5)

Documentation · Presets · IP Location · Import XPS Settings

Page Navigator

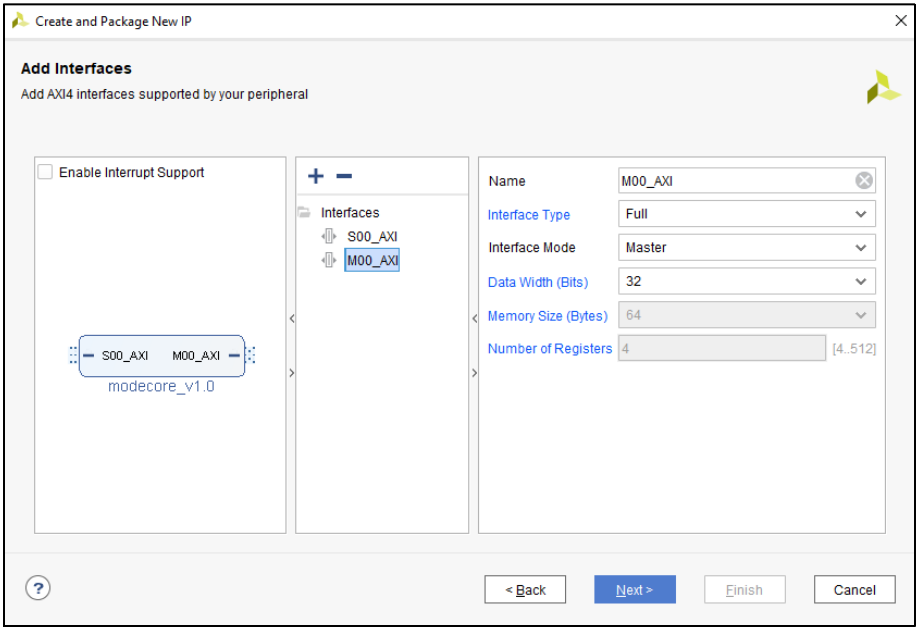
- Zynq Block Design
- PS-PL Configuration
- Peripheral I/O Pins
- MIO Configuration
- Clock Configuration
- DDR Configuration
- SMC Timing Calculation
- Interrupts

Interrupts Summary Report

Search:

Interrupt Port	ID	Description
☑ Fabric Interrupts		Enable PL Interrupts to PS and vice versa
PL-PS Interrupt Ports		
☑ IRQ_F2P[15:0]	[91:84], [6...	Enables 16-bit shared interrupt port from the PL. MSB is assigned th...
□ Core0_nFIQ	28	Enables fast private interrupt signal for CPU0 from the PL
□ Core0_nIRQ	31	Enables private interrupt signal for CPU0 from the PL
□ Core1_nFIQ	28	Enables fast private interrupt signal for CPU1 from the PL
□ Core1_nIRQ	31	Enables private interrupt signal for CPU1 from the PL
PS-PL Interrupt Ports		

Figure 7-8: ZYNQ7 block customization – adding PL to PS interrupt



Create and Package New IP

Add Interfaces

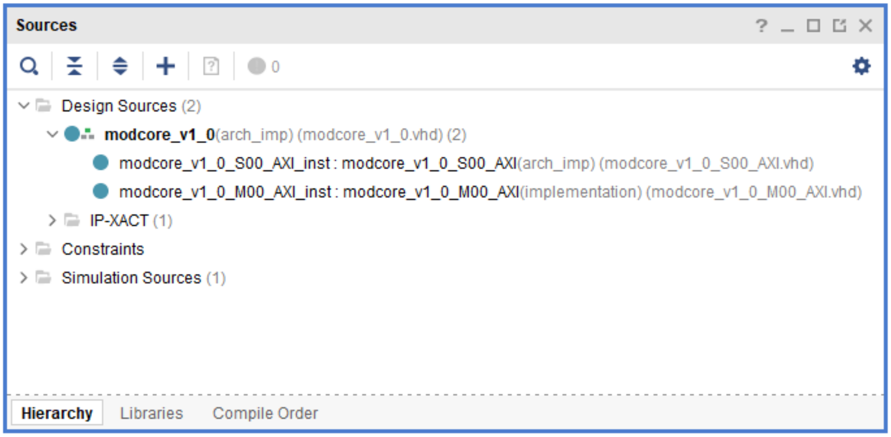
Add AXI4 interfaces supported by your peripheral

□ Enable Interrupt Support

Interfaces
- S00_AXI
- M00_AXI

Name	M00_AXI
Interface Type	Full
Interface Mode	Master
Data Width (Bits)	32
Memory Size (Bytes)	64
Number of Registers	4 [4..512]

Figure 7-9: Create and Package a Modcore coprocessor



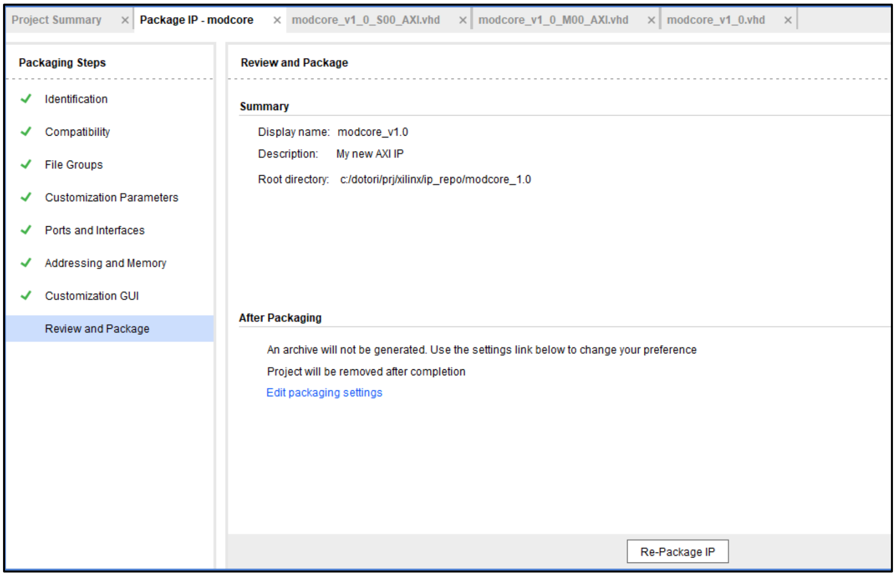
```
Design Sources (2)
  modcore_v1_0(arch_imp) (modcore_v1_0.vhd) (2)
    modcore_v1_0_S00_AXI_inst : modcore_v1_0_S00_AXI(arch_imp) (modcore_v1_0_S00_AXI.vhd)
    modcore_v1_0_M00_AXI_inst : modcore_v1_0_M00_AXI(implementation) (modcore_v1_0_M00_AXI.vhd)
  IP-XACT (1)
Constraints
Simulation Sources (1)
```

Figure 7-10: Source files for three basic modules in Modcore coprocessor

b) *modcore_v1_0_S00_AXI.vhd*
Modcore AXI slave module. This module will work with AXI master interface in PS subsystem. DMA device driver in PS system writes or reads the registers in Modcore AXI slave module.

c) *modcore_v1_0_M00_AXI.vhd*
Modcore AXI master module. This module will work with AXI slave module in PS system. This module work with AXI slave interface in PS subsystem which is added in step 11. The DMA module in Modcore AXI master module uses this interface to access the RAM memory.

15) Open each file in Vivado Sources window and overwrite it with the same file in *genesis/ip_repo/modecore_1.0/hdl*. Don't copy the file itself and paste it into *ip_repo/modcore_1.0/hdl* in the Vivado project.
16) click "Package IP". In the Packaging Steps, go to each item and click the link line of each item to make it checked as figure 7-11.
17) Go to Review and Package -> click "Re-Package IP". Vivado will close this IP edit window. At this step, we have created a Modcore IP and added it into Vivado's IP repository.
18) Now, let's add the Modcore coprocessor IP into our base project. In Vivado's Block Diagram window, click "+" button and type "modcore" in the search box. It should display our Modcore hardware IP. Select it. After this, Vivado will display "Run Connection Automation". Click it. Then check all checkbox by default values as below figure 7-12. Vivado will automatically connect AXI interface modules to its counter part modules.



Packaging Steps

- ✓ Identification
- ✓ Compatibility
- ✓ File Groups
- ✓ Customization Parameters
- ✓ Ports and Interfaces
- ✓ Addressing and Memory
- ✓ Customization GUI
- Review and Package

Review and Package

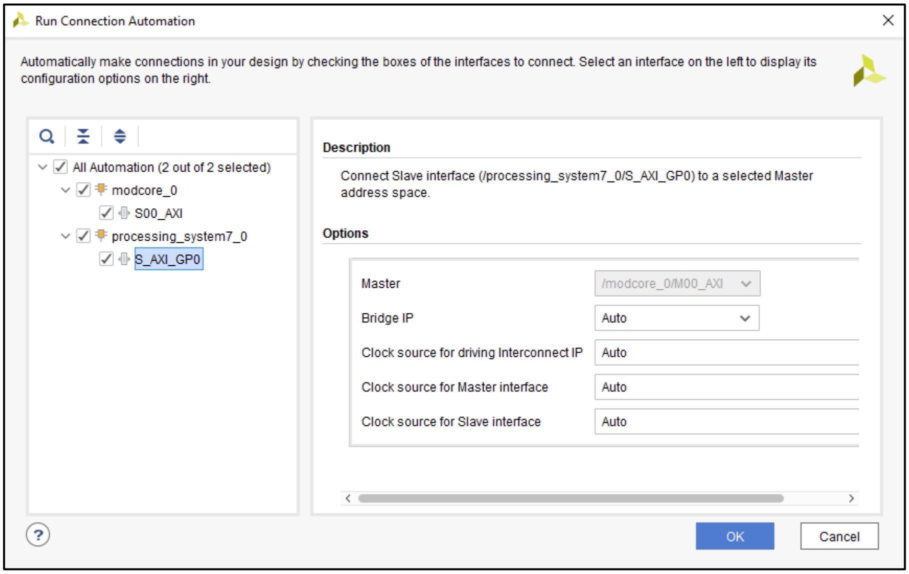
Summary

Display name: modcore_v1.0
Description: My new AXI IP
Root directory: c:/dotori/prj/xilinx/ip_repo/modcore_1.0

After Packaging

An archive will not be generated. Use the settings link below to change your preference
Project will be removed after completion
Edit packaging settings

Figure 7-11: Package Modcore coprocessor IP



Automatically make connections in your design by checking the boxes of the interfaces to connect. Select an interface on the left to display its configuration options on the right.

- ☑ All Automation (2 out of 2 selected)
  - ☑ modcore_0
    - ☑ S00_AXI
  - ☑ processing_system7_0
    - ☑ S_AXI_GP0

Description

Connect Slave interface (/processing_system7_0/S_AXI_GP0) to a selected Master address space.

Options

Master	/modcore_0/M00_AXI
Bridge IP	Auto
Clock source for driving Interconnect IP	Auto
Clock source for Master interface	Auto
Clock source for Slave interface	Auto

Figure 7-12: Automated connection between Modcore and PS subsystem

19) Connect the SW_DMA_IRQ signal in Modcore to the IRQ_F2P pin in ZYNQ. This is easily done by mouse left click on one of those pins and then drag mouse to the other pin. This signal is used for interrupt from Modcore to PS subsystem. The final block diagram looks like below. Figure 7-13 has Modcore processor, ZYNQ processor blocks,

all AXI master-slave interfaces are connected and a new interrupt signal pin is connected.

20) Then, create the top view of our block diagram. Click a design_1 in Sources and select "Create HDL Wrapper…" as figure 7-14.

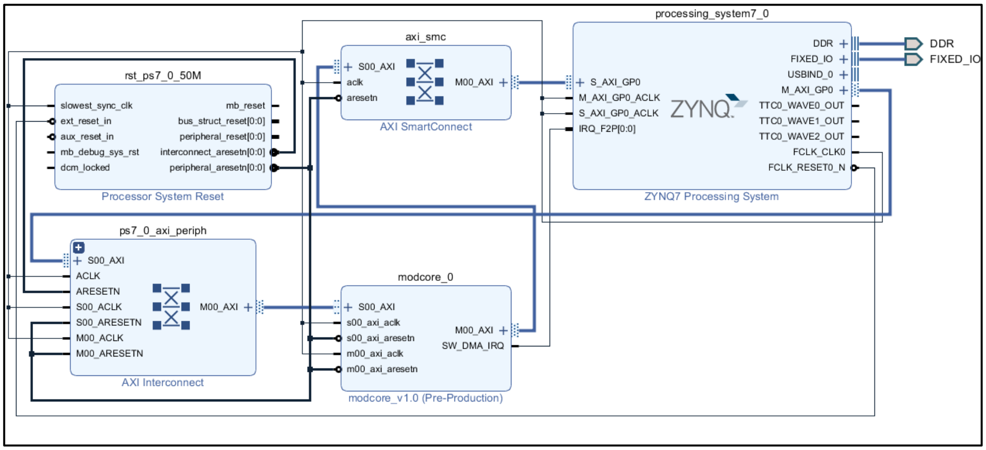


Figure 7-13: Final block design with Modcore coprocessor and ZYNQ processor

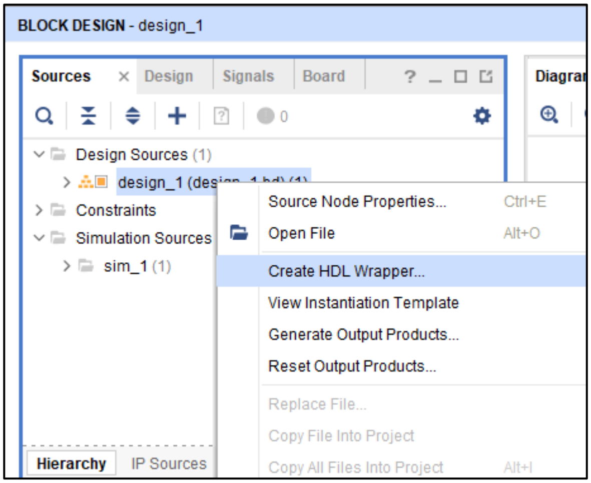


Figure 7-14: Create an HDL Wrapper for top view of design

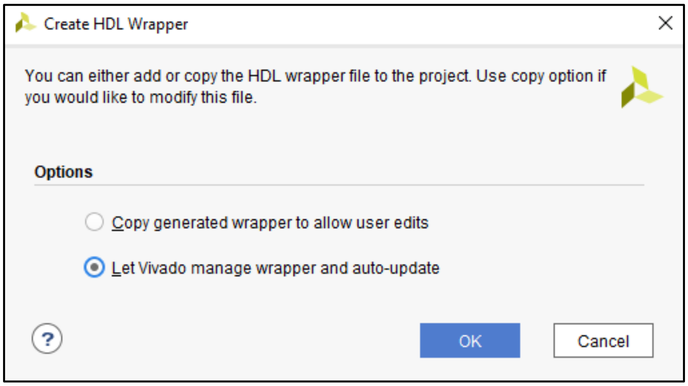
21) check "Let Vivado manage wrapper and auto-update" as in figure 7-15.

22) Now, it's time to build our bitstream that includes modcore IP coprocessor. Click "generate bitstream" in Flow Navigator.

23) After a while, the bitstream must be generated successfully. Go to File -> Export ->

Export hardware... and check “Include bitstream”.

24) Copy *design_1_wrapper.bit* bitstream into the petalinux build directory described in chapter 2.3.7 and build BOOT.BIN with a *petalinux-package* command. Copy this BOOT.BIN into the SD card of target board, then try to boot with this new bitstream.


Create HDL Wrapper

You can either add or copy the HDL wrapper file to the project. Use copy option if you would like to modify this file.

Options

○ Copy generated wrapper to allow user edits

● Let Vivado manage wrapper and auto-update

Figure 7-15: Select Vivado for top-level wrapper

7.3 DMA Device Driver in SLOS

High level OS like Linux has a well-defined its device driver framework. This framework helps hardware manufacturers to make their device driver cooperate with kernel core (e.g. scheduler, power management, interrupt). Engineers from different companies can develop a device driver for their custom hardware by complying with this predefined device driver framework.

SLOS doesn’t have any device driver framework. Instead, we will demonstrate what is general knowledge and what is needed for a device driver development. Most device driver has the abilities to program the registers of the hardware and to handle the interrupt from the hardware. On top of these core abilities, device driver framework of high-level OS can have a fancy functionalties such as hierarchy of device drivers to handle complex hardware (e.g., USB), power management and so on. As the name represents, SLOS has a simple DMA device driver which programs the registers in AXI slave module and handles the interrupt from Modcore processor.

7.3.1 Source Changes in SLOS

DMA device driver sources are newly added into *drivers/dma* directory. The *Makefile* is also changed to build files in *drivers/dma/* directory. Following is the summary of SLOS source changes from chapter 6 to implement the DMA device driver.

1) *kernel/drivers/dma/dma.c, kernel/inc/dma.h*

This file has the source code for DMA device driver implementation. Below are the basic functions in *dma.c.*

a) *void init_dma(void)*
   This function has the initialization of DMA hardware.

b) void set_dma_work(uint32_t src, uint32_t dst, uint32_t len)
   This function sets the DMA work order. *uint32_t src* is an address of source memory. *uint32_t dst* is the address of destination memory. *uint32_t len* is the byte length of DMA operation.

c) int start_dma(void)
   This function starts the DMA transaction between AXI master of Modcore processor and RAM memory.

d) *int dma_irq(void *arg)*
   This function is the DMA interrupt handler. DMA device driver can figure out the right time of Modcore processor's job completion and it can prepare next job or apply some other application logic to the results of Modcore processor.

2) Makefile
   The Makefile is changed to build the DMA source files in *driver/dma* directory. This is done by adding a new build directory path (*driver/dma)* to the variable *$(KERNMODULES)*.

3) *kernel/core/main.c*
   The *start_kernel()* is changed to be added a DMA hardware initialization function call during boot up.

4) *kernel/core/task.c*
   This file is changed to be added a *cfs_worker3* task for the demonstration of DMA work. This demo work does a one-time job to start a DMA work. The *cfs_worker3* task sends a work order that moves 64Kbytes of data from 0x2000_0000 to 0x3000_0000. While moving the memory data, Modcore coprocessor hardware performs a simple 256's complement operation.

7.3.2 Initialization of Mod Coprocessor

The initialization of Modcore processor is very simple because the Modcore processor hardware doesn't have any power lines connected, doesn't have any boot-up sequences. Generally, the hardware needs power and boot sequence for its initialization. But the initializatioin process of Modcore processor has none of these and has only the interrupt initialization for PL to PS interrupt.

Before this chapter 7, we have been using only one interrupt; a timer interrupt. The void init_dma(void) function registers a new DMA interrupt signal into the GIC controller. This function calls two GIC functions to register a new interrupt signal.

1) *gic_register_int_handler(DMA_IRQ_ID, dma_irq, NULL)*
   This function registers a new interrupt handler into a GIC interrupt handler list. The *DMA_IRQ_ID* is defined as 61 in *kernel/inc/dma.h.* This number is interrupt number pre-assigned by Zynq chipset. This was covered in chapter 4.4.5. The *dma_irq* is a function pointer to DMA interrupt handler function.

2) *gic_mask_interrupt(DMA_IRQ_ID)*
   This function enables the DMA interrupt in the GIC controller. This function sets the corresponding bits in GIC_ICDISER register, then the GIC controller will forward the DMA interrupt coming from PL subsystem to the CPU interface. The GIC will call proper interrupt handler registered in its IRQ handler vector table.

The DMA initialization can be done in any place during the system bootup. In some cases, the order of device initialization sequence is important. For example, bus must be initialized first and then each device attached to that bus can be initialized. But in case of DMA device driver in SLOS, the initialization order isn't that important. Just add the init_dma() function before *cpuidle()* in *start_kernel()* function.

7.3.3 Set DMA Work Order

The Modcore DMA module gets one work order at a time. That work order has a source address, destination address, and byte length of data and can be set by *set*_dma_work() function. The device driver can implement scatter-gather DMA operation by using the set_dma_work() function and the interrupt from Modcore coprocessor. By listing up the work orders, it can program the Modcore coprocessor with a different source address, length and destination address at a time. This different work orders can build the *scatter-gather* in DMA operation. Below is the DMA work order generation function.

```
void set_dma_work(uint32_t src, uint32_t dst, uint32_t len)
{
    int i, q, r;
    struct dma_work_order *pcur, *ptemp;

    if (len > DMA_BURST_LEN) {
        q = (int)(len / DMA_BURST_LEN);
        r = len % DMA_BURST_LEN;

        for (i = 0; i < q; i++) {
            ptemp = (struct dma_work_order *)kmalloc(sizeof(struct dma_work_order));
            ptemp->order_num = i;
            ptemp->src = src + DMA_BURST_LEN * i;
            ptemp->dst = dst + DMA_BURST_LEN * i;
            ptemp->len = DMA_BURST_LEN;
            ptemp->next = NULL;
            if (i == 0) {
                p_dma_work_order = pcur = ptemp;
            } else {
                pcur->next = ptemp;
                pcur = pcur->next;
            }
        }

        if (r) {
            ptemp = (struct dma_work_order *)kmalloc(sizeof(struct dma_work_order));
            ptemp->order_num = i;
            ptemp->src = src + DMA_BURST_LEN * i;
            ptemp->dst = dst + DMA_BURST_LEN * i;
            ptemp->len = r;
            ptemp->next = NULL;

            if (pcur) {
                pcur->next = ptemp;
            }
        }
    } else {
        ptemp = (struct dma_work_order *)kmalloc(sizeof(struct dma_work_order));
        ptemp->order_num = 0;
        ptemp->src = src;
        ptemp->dst = dst;
```

```
42              ptemp->len = DMA_BURST_LEN;
43              ptemp->next = NULL;
44
45              p_dma_work_order = ptemp;
46          }
47      }
```

This simple routine builds work list by splitting the input length with a *DMA_BURST_LEN* which is defined as just one memory frame size (4KB). Line 7 ~ 8 calculates the quotient and remainder of input data length. After spliting, the *set_dma_work()* function creates a linked list of these one memory frame size work orders. Line 6 checks the input length and if it is bigger than *DMA_BURST_LEN*, then the block from line 7 to 40 allocates a *dma_work_orders* and builds a linked list. These routines seem to be straight forward.

If the input length is smaller than *DMA_BURST_LEN*, there is only one work order. Since current SLOS has a limited number of virtual memory region descriptor (around 170 region descriptors are allowed) for its heap, the input length should not be bigger than 170 * *DMA_BURST_LEN*. Actually, there are other places which calls the kmalloc() in SLOS, the possible number of work order is a little bit smaller than this number. If you want to transfer bigger memory region through Modcore DMA module, you can redefine the *DMA_BURST_LEN* in *kernel/in/dma.h* with a larger number. Line 41 ~ 50 is for allocating a single worker order in the heap.

As mentioned before, with this linked list of DMA work order, we can build *scatter-gather* DMA operation. We build a linked list of work orders with a scattered memory region (non-contiguous memory region) for the source memory region, with a gathered memory region (contiguous memory region) for the target memory region then feed those work orders one by one whenever the Modcore processor generates an interrupt. The interrupt occurs only when current work order is completed. But each work order in the work order list still covers a contiguous memory region.

7.3.4 Start Modcore operation and Cache Coherency

For starting the operation of Modcore processor, set the control register bit[0]. This triggers the AXI master's DMA module in Modcore processor to operate without any PS subsystem involvement.

In DMA operation, the *cache coherency* problem between ARM processor in PS subsystem and Modcore coprocessor in PL subsystem rises. For example, the *cfs_worker3* task in PS updates the memory data and makes the Modcore read the memory data. While the *cfs_worker3* writes the data to memory, the last few kilo-bytes could be stay in cache memory. After this, if the DMA module read the data in the memory, the memory data would be old data, not yet written by *cfs_worker3* task. To solve this, normally a cache flushing operation must be done before triggering the DMA module to read the correct memory data. Nonetheless, if you remember the page property settings in chapter 5, all pages in SLOS is set as *non-cacheable* even in *normal* memory type. So actually, SLOS doesn't have a cache coherency problem between ARM processor in PS and Modcore coprocessor in PL. But let's have a quick look at on the cache coherency happening in general use case of DMA operations.

Zynq chipset memory interface looks like figure 7-16. Zynq chipset has L1 and L2 caches which are tightly coupled with Cortex-A9 ARM processor. L1 caches is made up of separate 32KB data cache (*D-Cache)* and 32KB instruction cache *(I-Cache)*. Since L1 cache is tightly coupled with the processor and the Cortex-A9 processor has dual core processors, the L1 caches between processors need to be coherent each other. The *Snoop Controll Unit (SCU) automatically* takes care of the L1 cache coherency between two processors without software intervention. SCU can copy clean data and move dirty data directly between participating L1 caches, without having to access (and wait for) external memory. L2 cache is 512KB and it is an instruction and data unified cache. After L2 cache, the memory interface is connected to main external memory.

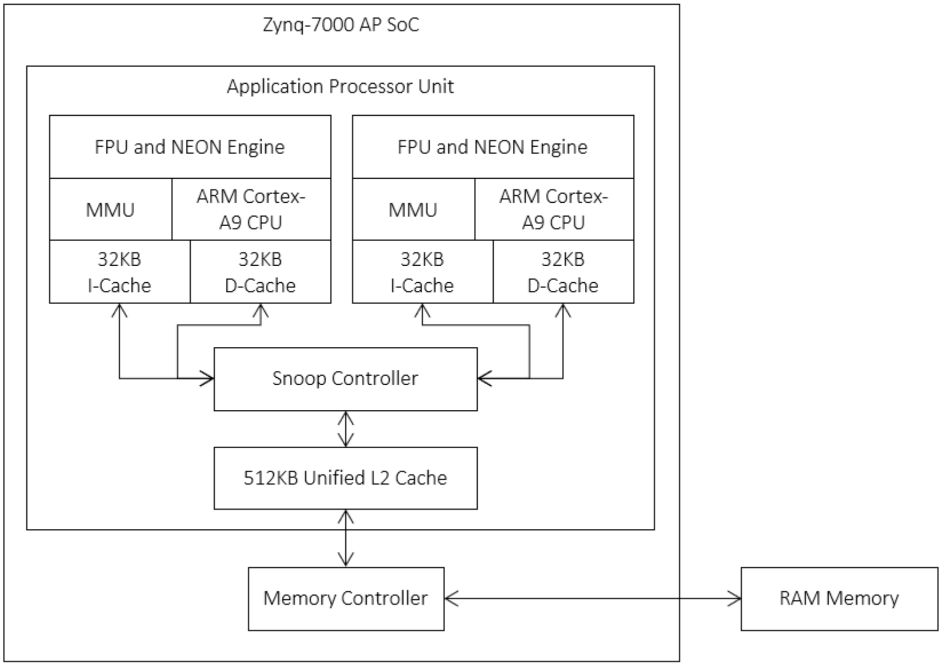


Figure 7-16: Zynq chipset memory interface hierarchy

Since these caches are used by ARM processor only, the Modcore processor in PL subsystem fails to catch the changes stored in these caches by SLOS which is running in the ARM processor. Coherency is about ensuring all processors, or bus masters in the system see the same view of memory. It means that changes to data held in the cache of one core are visible to the other cores, making it impossible for cores to see stale copies of data (the old data from before it was changed by the first core). For example, if you have a processor that is creating a data structure then passing it to a DMA engine to move, both the processor and DMA must see the same data. If that data were cached in the core and the DMA reads from external memory, the DMA will read old, stale data [6]. Since the Modcore coprocessor is directly access to the meory without the ARM processor involvement, we should make all modules (ARM processors and Modcore coprocessor) see the coherent content of the main memory. If we work on software running only in the ARM processor, the Snoop controller for L1 cache and L2 cache controller make sure the cache coherency. Cache coherency is maintained by hardware and we don't need to sweat on caches in this case. But now, it is ARM processor that accesses the cache and the DMA module in Modcore coprocessor needs to see the coherent memory data with the ARM processor. For this, the device driver must make all data in the cache flush into the main memory before starting the DMA operation so that the following Modcore operation reads the correct data values written from SLOS.

Below, I cited the basic idea on how to maintain cache coherency in ARM from ARM Cortex-A Programmer's Guide [6].

Normally, there are three mechanisms to maintain coherency:

1) *Disable caching*

   This is the simplest mechanism but might cost significant core performance. To get the highest performance processors are pipelined to run fast, and to run from caches that offer a very low latency. Caching of data that is accessed multiple times increases performance significantly and reduces DRAM accesses and power. Marking data as "non-cached" could impact performance and power.

2) *Software managed coherency*

   Software managed coherency is the traditional solution to the data sharing problem. Here the software, usually device drivers, must clean or flush dirty data from caches, and invalidate old data to enable sharing with other processors or masters in the system. This takes processor cycles, bus bandwidth, and power. Where there are high rates of sharing between requesters the cost of software cache maintenance can be significant, and can limit performance.

3) *Hardware managed coherency*

   Hardware Coherency is the most efficient solution. Any data marked 'shared' in a hardware coherent system will always be up to date. All cores and bus masters in that sharing domain see the exact same value. While hardware coherency might add some complexity to the interconnect and clusters, it greatly simplifies the software and enables applications that would not be possible with software coherency.

There are a number of standard ways by which cache coherency schemes can operate [6]. Most ARM processors use the *MOESI* protocol, while the Cortex-A9 uses the *MESI* protocol. Depending on which protocol is in use, the SCU marks each line in the cache with one of the following attributes: *M (Modified), O (Owned), E (Exclusive), S (Shared) or I (Invalid).* These are described below:

1) *Modified*
   The most up-to-date version of the cache line is within this cache. No other copies of the memory location exist within other caches. The contents of the cache line are no longer coherent with main memory.

2) *Owned*
   This describes a line that is dirty and in possibly more than one cache. A cache line in the owned state holds the most recent, correct copy of the data. Only one core can hold the data in the owned state. The other cores can hold the data in the shared state.

3) *Exclusive*
   The cache line is present in this cache and coherent with main memory. No other copies of the memory location exist within other caches.

4) *Shared*
   The cache line is present in this cache and coherent with main memory. Copies of it can also exist in other caches in the coherency scheme.

5) *Invalid*
   The cache line is invalid.

These cache line state is summarized as below table.

	Dirtiness	Shared-ness
Modified	Yes	No
Owned	Yes	Yes
Exclusive	No	No
Shared	No	Yes

Table 7-3: Summary of cache line state based on Dirtiness and Shared-ness

The rules for the standard implementation of the protocol are as follows [6]:

1) A write can only be done if the cache line is in the Modified or Exclusive state. If it is in the Shared state, all other cached copies must be invalidated first. A write moves the line into the Modified State.

2) A cache can discard a Shared line at any time, changing to the Invalid state. A Modified line is written back first.

3) If a cache holds a line in the Modified state, reads from other caches in the system will get the updated data from the cache. Conventionally, this is done by first writing the data to main memory and then changing the cache line to the Shared state, before performing a read.

4) A cache that has a line in the Exclusive state must move the line to the Shared state when another cache reads that line.

5) The Shared state might not be precise. If one cache discards a Shared line, another cache might not be aware that it could now move the line to Exclusive status.

This knowledge is needed for developing a cache controller. In SLOS, there is a flush_ent_dache() function implementation for flush entire data cache in *kernel/core/cache.S,* which means we use a software managed cache coherency . Nonetheless, all memory in SLOS is not cacheable, we don't need to use this funtion in the DMA test.

Now, let's review important terms and concept regarding the cache coherency. These terms are coming from ARM Reference Manual [5].

1) Clean

A cache clean operation ensures that the updates made by an observer that controls the cache are made visible to other observers that can access the memory at the point to which the operation is performed.

2) Invalidate

A cache invalidate operation ensures that updates made visible by observers that access memory at the point to which the invalidate is defined are made visible to an observer that controls the cache.

3) Point of Coherency (PoC)

The PoC is the Point of Coherency which all agents that can access memory are guaranteed to see the same copy of a memory location. In many cases, this is effectively the main system memory.

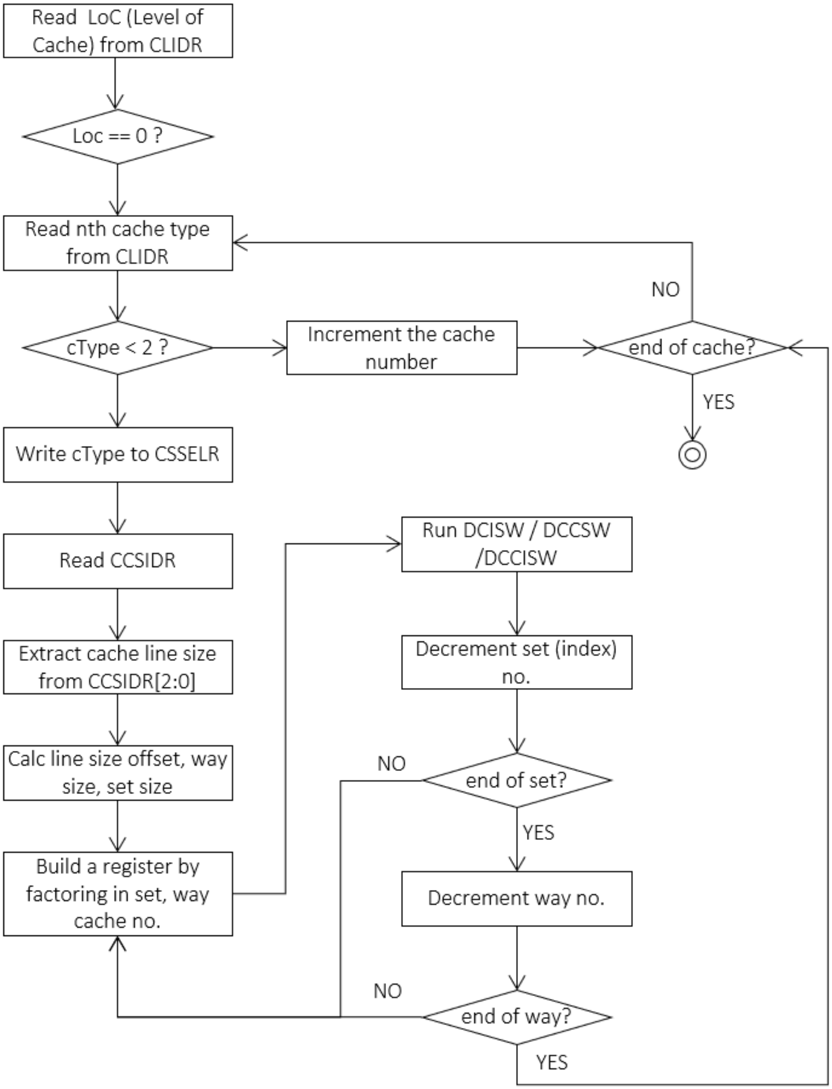

Read LoC (Level of Cache) from CLIDR
Loc == 0 ?
Read nth cache type from CLIDR
cType < 2 ?
Increment the cache number
end of cache?
NO
YES
Write cType to CSSELR
Read CCSIDR
Extract cache line size from CCSIDR[2:0]
Calc line size offset, way size, set size
Build a register by factoring in set, way cache no.
Run DCISW / DCCSW /DCCISW
Decrement set (index) no.
end of set?
NO
YES
Decrement way no.
end of way?
NO
YES

Figure 7-17: Cache flushing procedure flow

4) *Point of Unification (PoU)*
The PoU for a processor is the point by which the instruction and the data caches and the translation table walks of that processor are guaranteed to see the same copy of memory location. In many cases, the point of unification is the point in a uniprocessor memory system by which the instruction cache and data caches and the translation table walks have merged.

5) *Level of Coherency (LoC)*

   This is a field in CLIDR (Cache Level ID Register). This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of coherency.

Cache flushing is a process used to flush data of ARM prcoessor to the main memory (PoC). The implementation in the flush_ent_dache() function gets the Level of Coherency (*LoC)* value from CLIDR register and runs a clean command by *way/set* for all cache levels. A detailed description is skipped here but the procedure of flush_ent_dcache() is depicted in figure 7-17. You can see there are two loops that runs cache management commands (DCISW, DCCSW, DCCISW) until the end of way/set.

7.3.5 DMA Interrupt Handler

Modcore coprocessor generates an interrupt whenever it completes current work order. The interrupt ID used by Modcore coprocessor is 61. This is one of PL to PS interrupts. One interrupt line from Modcore in PL subsystem is connected to the PS subsystem as in figure 7-13 (We will connect another interrupt line to PS in next chapter). The interrupt polarity of PL to PS is set as a *level- triggered* by default. When the Modcore raises this signal to high, it must keep this signal high until the PS interrupt controller detects it. When the DMA device driver's interrupt handler is called, it first needs to acknowledge the interrupt. After DMA device driver finishes its job, it should set the *INTR_DONE* bit in the *control register* (bit[1]). Then, the AXI master module in Modcore coprocessor catches this and lowers the interrupt line.

When PS subsystem detects the SW_DMA_IRQ signal is high, this event goes through the GIC distributor and GIC CPU interface as described in chapter 4.4, and finally goes to the IRQ exception vector. The gic_register_int_handler() function called by init_DMA() function registers DMA interrupt service routine into the GIC's interrupt service vector list. The IRQ handler figures out the interrupt ID number and calls this DMA interrupt service routine from the handler's vector list. The DMA interrupt service routine reprograms the Modcore coprocessor with the next work order if there are remaining work orders in the job list. This DMA interrupt sequence is summarized in figure 7-18.

Since the IRQ exception handler disables interrupt and re-enable it when it returns, this interrupt service routine must finish its work very quickly. SLOS interrupt doesn't support re-enterant interrupt. This means while current interrupt is under processing, another interrupt can't kick in. If some routine in the interrupt handler takes too long, a timer interrupt could be lost and some time critical task could miss its deadline. So, we should avoid a heavy task that takes long in the interrupt service routine. Instead, we can partition the work in interrupt service routine, we can create a new oneshot task for the time-consuming portion and add it to the CFS scheduler's runqueue. After adding this task into the runqueue, the interrupt handler can return quickly and is able to respond to next interrupt before deadline. The task in the runqueue which is not time critical is scheduled by a CFS scheduler and is completed sometime in the future. We call the part in the interrupt handler which is time critical but doesn't take too long as a *Top Half* of interrupt. To the contrary, the other part which is scheduled later and non time critical part is called as a *Bottom Half*. We use a oneshot timer for the bottom half in this chapter, but in the next chapter, we will use a special CFS workq worker task that queues up functions and runs them. That is a better way but let's do that in the next chapter.

The timer interrupt handler which is composed of timer framework and scheduler's context swith is considered as Top Half and each task's meaningful running corresponds to a Bottom Half.

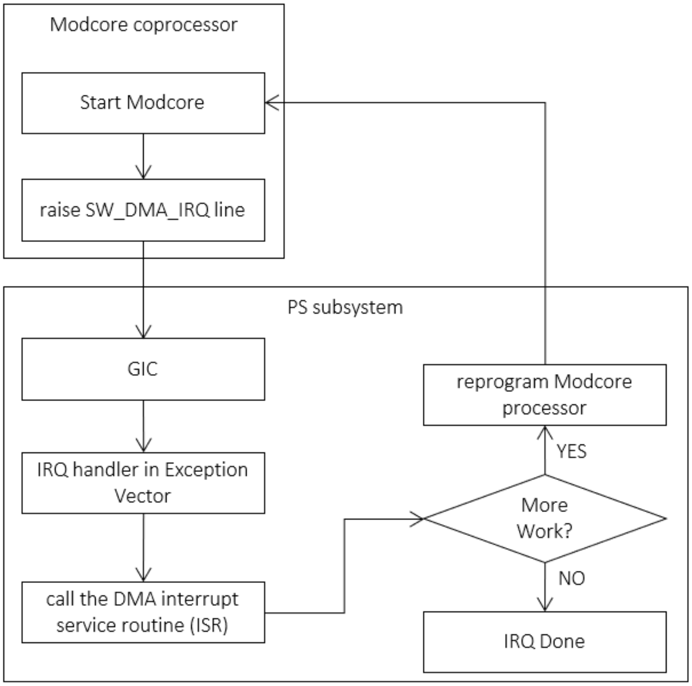

Modcore coprocessor
Start Modcore
raise SW_DMA_IRQ line
PS subsystem
GIC
IRQ handler in Exception Vector
call the DMA interrupt service routine (ISR)
reprogram Modcore processor
YES
More Work?
NO
IRQ Done

Figure 7-18: DMA interrupt process seqeuence

The dma_irq() function which is a DMA interrupt service routine is summarized below.

```
int dma_irq (void *arg)
{
```

```
3           uint32_t cntl;
4
5           cntl = readl(DMA_REG_CNTL);
6           cntl |= DMA_IRQ_DONE;
7           writel(cntl, DMA_REG_CNTL);
8
9           /* start next dma order */
10          if (start_dma()) {
11                  cntl = readl(DMA_REG_CNTL);
12                  cntl &= ~DMA_START;
13                  writel(cntl, DMA_REG_CNTL);
14
15                  xil_printf("dma done!\n");
16          }
17
18          return 0;
19      }
```

It looks quite straight-forward. Line 5 ~ 7 reads control register of AXI slave and sets the interrupt done bit and rewrite it to the control register. The start_dma() function in line checks whether there are more works in the list. If there are still works to do, then the start_dma() function re-programs the Modcore and have it start next work in the list. If start_dma() succeed in starting a next work, it returns 0. If there is no more work, it returns 0. Line 11 ~ 15 completes all the jobs in the work list and printf a message into the screen.

7.4 Design an Outstream Device in PL Subsystem

This chapter covers another example for hardware-software codesign. The purpose of this device is that hardware takes the memory data out to the external perpheral device continuously and software keeps adding its new job to the hardware job queue (Itab, Information Table). This data path has two buffers to save the temporal data and there are underflow, overflow control in each buffer. In our case, there is no real hardware peripheral connected to physical pin of Zynq, we will add a module to simulate the data consuming at the end of this data path. You could change this by connecting an external LED blinking to the end of data path.

While streaming the data from memory data to the data consumer, each module checks the data consistency by using the sequence number of each block. Each block of data has its sequence number programmed by outstream task, and this sequence number is checked by each module in PL system. If the sequence number isn't correct, the outstream top module generates an interrupt to PS system and the AXI slave module updates its status register with cause of interrupt. Then, the interrupt service routine in SLOS outstream device driver stops the outstream device. Outstream interrupt service routine can figure out the interrupt reason by reading the status register of outstream device.

Figure 7-19 summarizes the top view of outstream device modules. It is composed of odev device driver, AXI slave, AXI master, itab, Rd buffer, Data consumer. AXI Slave has registers for software programming. Task and device driver running in PS access these registers for bidirection communication, mostly for control and status. Itab and Rd buffer are used for buffering a temporal data. Itab stores the metadata from software and Rd buffer stores the data from external memory read by AXI master. These two buffers need a flow control (overflow and underflow) with *request* and *valid* signals between producer and consumer. As mentioned earlier, the Data consumer module simulates the real data out-going to external peripheral. Rd buffer and Data consumer subsystem have a feature to check the sequence number of each transaction block for data consistency. If the sequence number isn't correct, i.e. doesn't increase by 1 from previous sequence number, the odev top module generates an interrupt to PS.

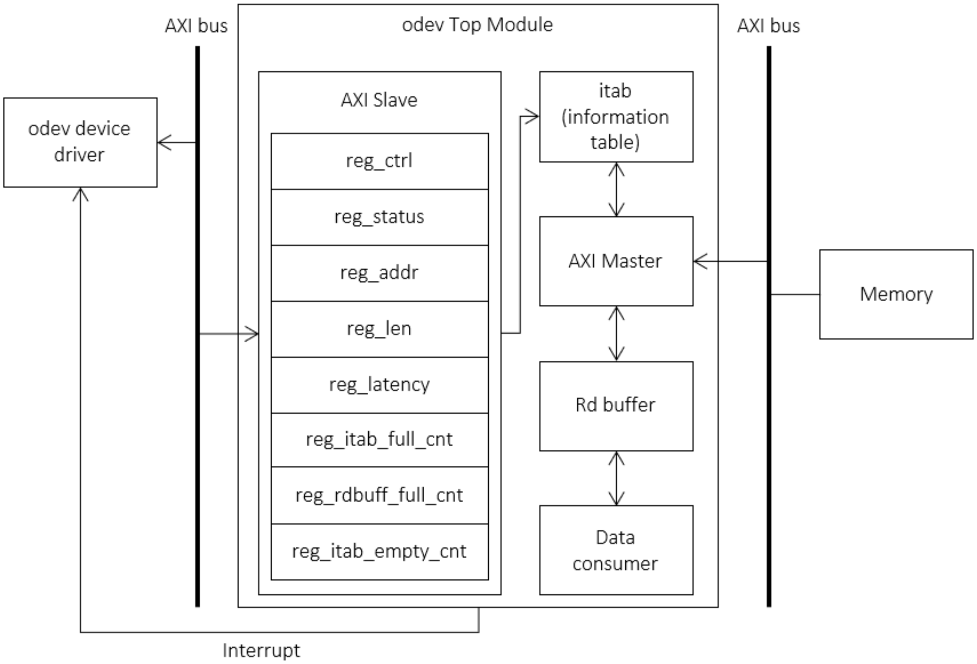


Figure 7-19: Top view for odev device driver, AXI Slave, AXI Master, itab, Rd buffer, Data consumer, AXI bus and Memory

The steps to add the Outstream device to Vivado isn't described because there is an automatic script in the latest SLOS branch (SLOS_CH7). These scripts will start Vivado, set up the project configuration and create the whole block design. All these steps are done in fully automatic way. You just need to generate a bitstream. But if you still do it by yourself, you can refer the steps in chapter 7.2.4.

7.4.1 Odev AXI Master

AXI master module plays a central role in Outstream device. It gets the metadata from Itab, reads data from memory and stores the read data to Rd buffer. Outstream AXI master uses the same burst mode as Modcore processor. It's burst length is 16 and beat size is 4 bytes, which means 64 bytes transfer per burst. Outstream device uses a block in each transfer which has 4 sequential bursts at a time. So, a block size is 256 bytes in total. Currently, the AXI master supports only 4 burst transactions for each Itab entry, i.e. the length for memory data is fixed as 256 per Itab entry. The first 4 bytes are used to store the sequence number of a current block by odev task and 255 bytes can be used to store the meaningful data. Figure 7-20 shows the components of a block.

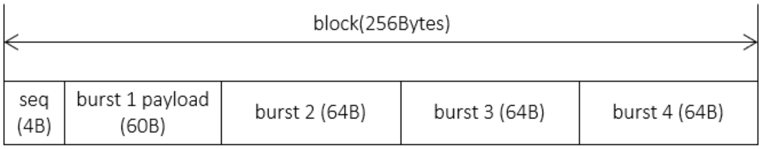


Figure 7-20: A block component in Outstream device

AXI master in Outstream device works as a metadata consumer with Itab and as data producer with Rd buffer and should follow the flow control in both buffers at once. Flow control prevent each module from overwrite previous entry or from pulling outdated entry. We will look into this in chapter 7.4.6. All these transactions are controlled by the state machine in AXI master.

Outstream AXI master's state machine looks like figure 7-21.

1) *IDLE*

   Reset all signals. All other states go into this state when odev device is stopped.

2) *ITAB_READ*

   When started, it is going to read the Itab entry for the first block. Triggering start is initiated by a task running in PS with the control register of AXI slave. This state asserts a request signal to Itab module and goes to next ITAB_READ_CHK state.

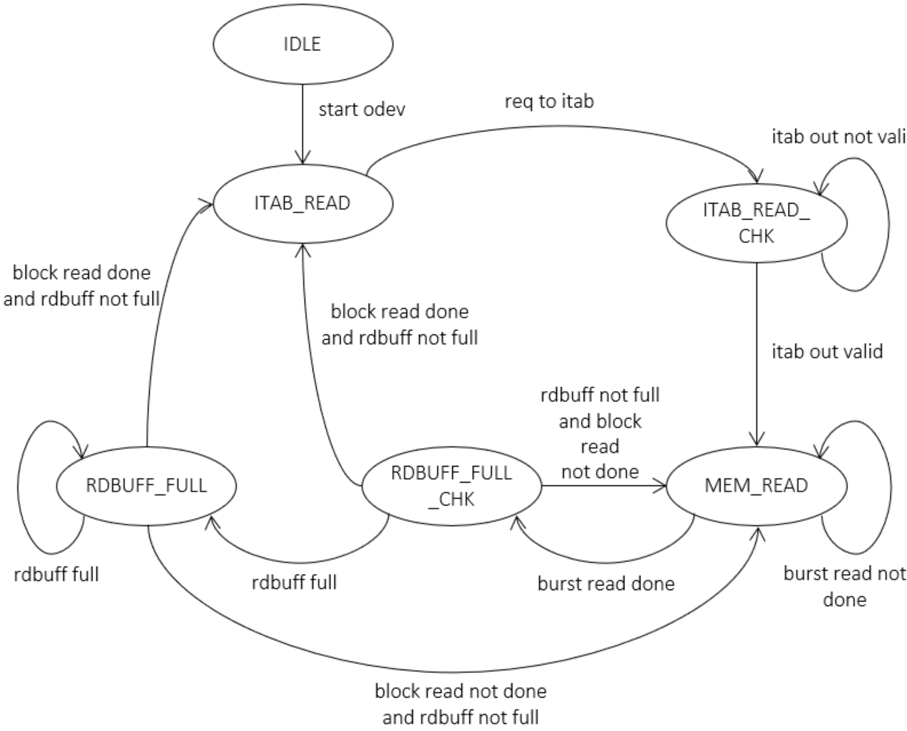


Figure 7-21: State machine of AXI master in Odev

3) *ITAB_READ_CHK*
This state waits until Itab module sends source address and length with the valid signal. If valid signal is not asserted, this state keeps staying in current state. If this stage gets valid metadata from Itab module, it goes to MEM_READ state.

4) MEM_READ
This state programs the AXI master protocol to read memory data. AXI burst type is 16 burst length, 4 bytes for burst size (beat size), and *INC* burst type are used. For details, refer AXI specification document [2]. When master module reads the memory data, it saves the data into *Rd_buff* module directly. When AXI master finishes one burst read, it goes to RDBUFF_FULL_CHK state to check the Rd_buff is full. Or current burst read is not done (one burst size is 64 bytes read, 16 beat transactions), stay in current state to read the rest beats from memory.

5) RDBUFF_FULL_CHK

Whenever AXI master completes one burst, it goes to this state. AXI master module should check Rd_buff full state and if Rd_buff is full, AXI master shouldn't start next burst read. Since the entry size in Rd_buff is for one burst size, this checkup is performed for every burst. If current Rd_buff state isn't full yet, this state goes back to MEM_READ state for next burst read. If current Rd_buff is full, this state goes to RDBUFF_FULL state.

6) *RDBUFF_FULL*
AXI master stays in this state until Rd_buff has a free space. When DataConsumer module take out some entries from Rd_buff, this state goes either to MEM_READ state when a block read is not finished or to ITAB_READ state when a block read is completed.

Unlike previous Modcore AXI master, Outstream AXI master uses only AXI read signals. It doesn't need to write the data back to the memory and quite simpler than Modcore AXI master. Currently, the block size (256 bytes), burst length, size are all fixed, but for higher performance these configuration could be changed.

7.4.2 Odev AXI Slave

AXI slave is a module that is directly interfaced to PS tasks. As we already did in Modcore device driver development, the registers are the keys for software to control, configure the hardware. Following table 7-4 describes the details of AXI slave registers. Base address for Outstream device is 0x43C0_0000.

Register Name	Address (Offset)	Width (bits)	Description
control	0x00000000	32	Control register
status	0x00000004	32	Status register.
src_addr	0x00000008	32	Source address register.
src_len	0x0000000C	32	Transaction length register. This is always 256.
con_latency	0x00000010	32	DataConsumer latency value register.

Table 7-4: Odev AXI Slave register description

Control register bits are explained in table 7-5.

Field Name	Bits	Type	Description
reserved	31:5	RW	Reserved

CONSUMER_START	4	RW	A bit to start or stop DataConsumer module alone.
STREAM_START	3	RW	A bit to start or stop streaming modules. These modules are the AXI master, Rd_buff and DataConsumer modules.
IN_TRANS	2	RW	This bit is used for a valid signal from software to Odev hardware (Itab).
INT_DONE	1	RW	Interrupt done bit. Odev device driver in PS must set this bit when Odev interrupt service routine completes its job and returns. Once this bit is set, Odev module lowers the physical interrupt signal to PS.
SLV_START	0	RW	This bit is used to start AXI slave and Itab module.

Table 7-5: Odev control register bit description

Status register bits are explained in table 7-6.

Field Name	Bits	Type	Description
reserved	31:6	R	Reserved
SEQ_ERR	5	R	This bit is set when sequence error occurs.
RDBUFF_FULL	4	R	This bit is set when Rd_buff is full.
IN_TRANS_DONE	3	R	This bit is set when sending current metadata is completed. Current metadata is stored in the Itab safely.
ITAB_FULL	2	R	This bit is set when Itab is full. Software should pause sending metadata when this bit is set.
RDBUFF_EMPTY	1	R	This bit is set when Rd_buff is empty.
ITAB_EMPTY	0	R	This bit is set when Itab is empty.

Table 7-6: Odev status register bit description

src_addr and *src_len* are 32bit registers for storing metadata of a block programmed by an Odev task. Since current Odev hardware supports only 256 bytes size for a block, src_len register is fixed as 256.

con_latency register is used to configure the speed of data consuming in the DataConsumer module. By changing this value, we can control make empty or full in the Itab and Rd_buff modules. If consuming speed is slow, the Rd_buff is first full and next the Itab becomes full. If consuming speed is fast, the Itab is first empty and Rd_buff finally becomes empty.

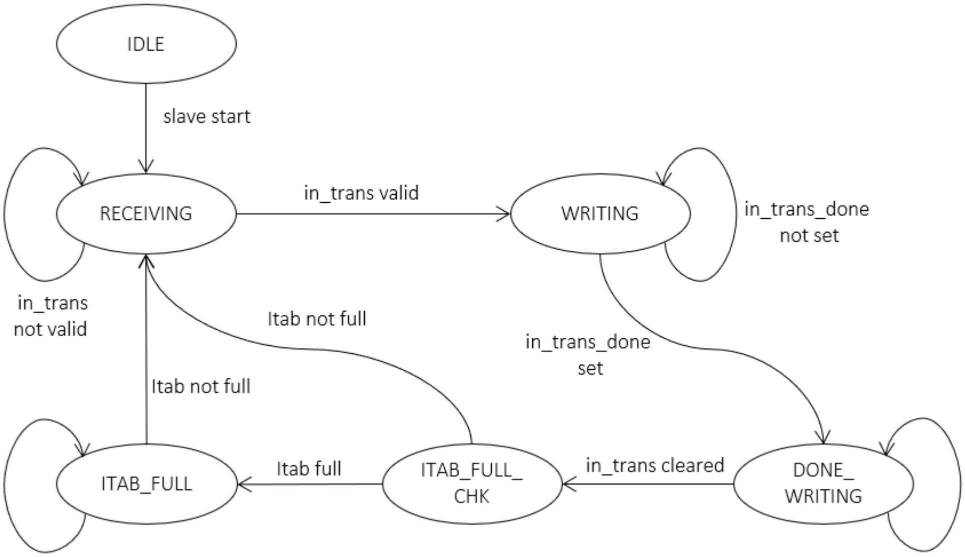


Figure 7-22: State machine of AXI slave in Odev

As in figure 7-22, AXI slave module also has its state machine to interface the Odev task in PS and Itab in PL.

1) *IDLE*
   Reset all signals.

2) *RECEIVING*
   Looping until the *in_trans* bit (control register bit[IN_TRANS]) is set. If this bit is set, it sends the src address, src length with valid signal to Itab module, and goes to WRITING state. Odev device driver asserts the in_trans bit in control register and then polling the *in_trans_done* status bit.

3) *WRITING*
   Looping until *in_trans_done* signal asserted by Itab module. When this signal is asserted, this state goes to DONE_WRITING state.

4) *DONE_WRITING*
   Set the in_trans_done status bit and looping until the *in_trans* cleared by Odev device driver. Odev device driver polls the in_trans_done bit in status register and clears the in_trans bit in control register. This 3-way hand-shaking closes the transaction to Itab entry. After finishing Itab entry transaction, this state goes to ITAB_FULL_CHK state.

5) *ITAB_FULL_CHK*

This state checks the Itab full status. If Itab isn't full, this state goes to RECEIVING state for next Itab entry transaction. If Itab is full, this state goes to ITAB_FULL state.

6) *ITAB_FULL*
This state is looping until there is a free space in Itab entry. If there are, this state goes to RECEIVING state to receive next Itab entry transaction.

7.4.3 ITAB Module

This module is a *block ram* table for the metadata (address, length) of a block. Itab entry is composed of 32 bit source address and 16 bit source length. So, each entry can support upto 64K size of a block, but in SLOS design, we fixed this size as 256Bytes. This information table has 512 entries in total. When it reaches at the end of the table entry, it wraps around to the first entry. There are two ways to generate block ram (*inferred block ram, using Xilinx block ram IP)*, and SLOS is using an inferred block ram generation. This method is quite straight forward and can design the custom signal controlling.

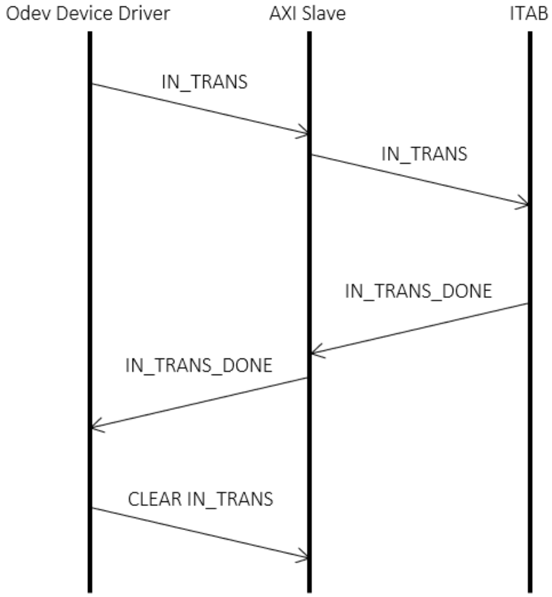


Figure 7-23: 3-way handshaking among Odev device driver, AXI Slave and Itab modules

Itab module gets the source address and length from AXI slave module and after it saves the metadata into one of its table entris, it asserts a signal (IN_TRANS_DONE) to AXI slave. AXI slave mode determines the time of saving input transaction with this signal and updates the status register bit[IN_TRANS_DONE]. Until Odev device driver clears the IN_TRANS bit of control register, the whole input transaction isn't completed. After the 3-way handshaking in figure 7-23 is completed, the input transaction is completed. After Odev device driver triggers the IN_TRANS transaction, it starts polling the status bit to send clear signal to finalize this transaction. This 3-way handshaking will be touched again in Odev device driver chapter.

Itab module is a buffer to save a temporal metadata until it is consumed by DataConsumer module. Streaming data with a buffering needs a flow control to prevent the overflow and overflow. Itab module controls the *empty* and *full* signals for this purpose.

Full signal goes to AXI slave module and empty signal goes to AXI master module. After current input metadata from device driver is saved to its last entry, Itab asserts full signal. When AXI slave gets this signal, it goes to its ITAB_FULL state and stays until there is free space in Itab. This prevents Itab overflow which overwrites the previous entry. If Itab is empty, Itab asserts an empty signal to AXI master. This prevents AXI master from pulling outdated entries.

The overflow or underflow in Itab flow control is solely dependent on the data sink module, DataConsumer module. If DataConsumer consumes the data slower than others, Itab will eventually experience its entry full state. This will pause the data input to Itab and give a little time for DataConsumer to make rooms in the buffers and the input transaction starts again. If DataConsumer is faster, Itab empty occurs and pause the DataConsumer until there are meaningful data in the buffers. While streaming the data to DataConsumer, this flow control happens quite often.

7.4.4 Rd Buffer Module

Rd_buffer module is a storage for data read from external memory. AXI master reads the memory data and saves it to Rd_buffer. Rd_buffer is another buffer generated by using inferred block ram. Rd_buffer entry size is 64Bytes which is exactly same as one burst size and there are 256 entries in Rd_buffer.

Rd_buffer is placed between AXI master module and DataConsumer module. As flow control in Itab, Rd_buffer has a flow control of empty, full. AXI master module checks Rd_buffer full after it copies current burst data to Rd_buffer. If it is full, then AXI_master doesn't read memory burst and wait until there is free space in Rd_buffer. This prevents AXI master from overwriting the previous data. DataConsumer module also waits the data read from Rd_buffer if there is no more valid burst data in it. This keeps DataConsumer from reading outdated data.

While saving burst data into its entry, Rd_buffer checks the sequence number stored in the first 4 bytes of the first burst in a block. If the sequence number isn't correct (skip or repeat number), then it asserts *RDBUFF_SEQ_ERR* signal to odev top module. Odev top module raise the interrupt signal to PS and the interrupt service routine in odev device driver is called. This is covered in chapter 4. Odev interrupt is another PL to PS interrupt, interrupt number is 62.

7.4.5 Data Consumer Module

This module consumes the burst data in Rd_buffer. It should not read the data when Rd_buffer is empty. Since this module is the end of the whole data path of outstream, it determines the outstream's speed, throughput. For example, if the consuming speed is slow, the Rd_buff will be full. This blocks AXI master's reading memory data, which results in Itab full. If Itab is full, the insert to Itab function *(put_to_itab)* in odev device driver doesn't return to caller task and keeps staying in polling the IN_TRANS_DONE bit. On the contrary, if the consuming speed is too fast, Rd_buff becomes empty soon, then DataConsumer should wait until there are valid data in Rd_buffer. AXI master, Itab AXI slave and outstream task in PS do their best to fill up the data in the streaming but DataConsumer module suffers from the starvation of empty Rd_buff.

DataConsumer module also checks the sequence number like Rd_buff and asserts a signal to the odev top module. Odev top module generates interrupt when sequence error occurs either in Rd_buff or DataConsumer.

7.5 Odev Task and Odev Device Driver

In this streaming device, once a stream task started, it should keep running to add new metadata to the Itab. For this, the odev task is created as one of CFS task and is scheduled by the CFS scheduler according to its priority. Odev task uses the functions odev device driver to initialize the hardware, start/stop the stream, and add a new entry into Itab.

Odev device driver supports the initialization of odev device, start / stop of sub modules (such as AXI master, DataConsumer), interrupt service routine, and adding Itab entry.

7.5.1 Source Changes in SLOS

New source files for odev device driver are added and some files are modified to support odev device. Those changes are almost same as dma device driver, and summarized as below.

1) *kernel/drivers/odev/odev.c, kernel/inc/odev.h*
   These files have the implementations of odev device driver.
2) *Makefile*
   Odev device driver source files in *driver/odev* directory are added to Makefile. This is simpley done by adding a new build directory path (*driver/odev)* to the variable *$(KERNMODULES)*.
3) *kernel/core/main.c*
   The *init_odev()* is added to the main entry for the initialization of odev during bootup.

4) *kernel/core/task.c*
   There is another CFS task created to issue new metadata work order to odev device.

As mentioned, these changes are just a copy of dma implementations except different memory mapped address and interrupt ID number.

7.5.2 Odev device driver

Odev device driver has following functions for application's usage.

1) *int32_t init_odev(void)*
   This function is used to initialize the odev device driver. It enables odev interrupt ID and registers its interrupt service routine handler to GIC. This was covered in chapter 7.3.2.

2) *int32_t start_odev(void), int32_t stop_odev(void)*
   This starts or stops the AXI slave and Itab modules. Start/stop AXI slave can be done through the SLV_START bit in control register.

3) *int32_t start_odev_stream(void), int32_t stop_odev_stream(void)*
   This starts or stops odev stream modules, those are AXI master, Rd_buff and DataConsumer modules. This is done through the STREAM_START bit in control register.

4) *int32_t start_consumer(void), int32_t stop_consumer(void)*
   This starts or stops the DataConsumer module alone. DataConsumer is placed at the end of data path and controls the streaming of data path. If it stops, Itab and Rd_buff buffers are all full state. The producer side of these buffer are suspending until DataConsumer module restarts.

5) *int32_t put_to_itab(uint32_t sAddr, uint32_t sLen)*
   This function is used to add a new entry into Itab. This function get the source address and length. Length is fixed as 256 bytes for now. This function follows the 3-way handshaking protocol with AXI slave and Itab modules as described in figure 7-22. This function spins forever and doesn't return to caller task until the IN_TRANS_DONE bit in the status register. Once IN_TRANS_DONE bit is set, then it clear the IN_TRANS bit of control register to make AXI slave move to next state (ITAB_FULL_CHK). If Itab is already full, this function returns positive integer. The caller function can check the success or failure with this return value. If current put_to_itab fails, the caller must retry the current entry.

6) *int odev_irq(void *arg)*
   This is an interrupt service routine for odev. Odev generates an interrupt when it finds an incorrect sequence number in the block. Each block has a unique sequence number and this can be used to verify the streaming data. Interrupt ID for odev is 62 which is defined in the odev.h file. When interrupt occurs, it stops the DataConsumer module by using stop_consumer().

7) *int32_t set_consume_latency(uint32_t lat)*
   This sets the speed of DataConsumer module. The unit of latency is a clock count of AXI bus clock.

7.5.3 Outstream Task

One of CFS task is assigned to Outstream task. This task starts the odev submodules sequentially, and calls push_to_itab() to add a new entry into Itab. This first starts the AXI slave and Itab by calling start_odev(). Next this prepares the block memory data. This is nothing but writing the sequence number into the first 4 bytes of block data. Since this task is for a test, Outstream task uses only 4 KB memory area and wrap around after the last block. Then, it sets the DataConsumer speed (set_consume_latency()) and starts the streaming modules by calling start_odev_stream(). Finally, this task loops forever calling put_to_itab() to add a stream work order to Itab. If put_to_itab() fails, this task retry current stream data.

7.6 Putting it altogether

To run the test, we need to create a test task. First, let's add a dma task into one of the CFS tasks. If you check out the latest master branch, the *kernel/core/task.c* file already has this change. Now, the *cfs_worker2* task runs a test for SLFS as described in the chapter 6. The *cfs_worker3* task runs a DMA test. The *cfs_worker3* task copies 0x10000 bytes from 0x2000_0000 to 0x3000_0000. The *DMA_BURST_LEN* is defined as 4KB (0x1000). Since there is a limit in the number of virtual memory pool descriptor as in chapter 5.6.2, if the copy data amount is too large, the kmalloc() in set_dma_work() will fail because of the shortage of virtual memory pool. In this test, the data amount is set as 0x10000 (64KB) that is equal to 16 DMA data transfer. At the end of each transfer, the Modcore coprocessor generates a DMA interrupt. The interrupt handler in DMA device driver can reprogram next DMA work. While copying, it runs a 256's complement operation. Run below command in the SLOS Git repository to checkout the latest sources for chapter 7.

```
git checkout SLOS_CH7
```

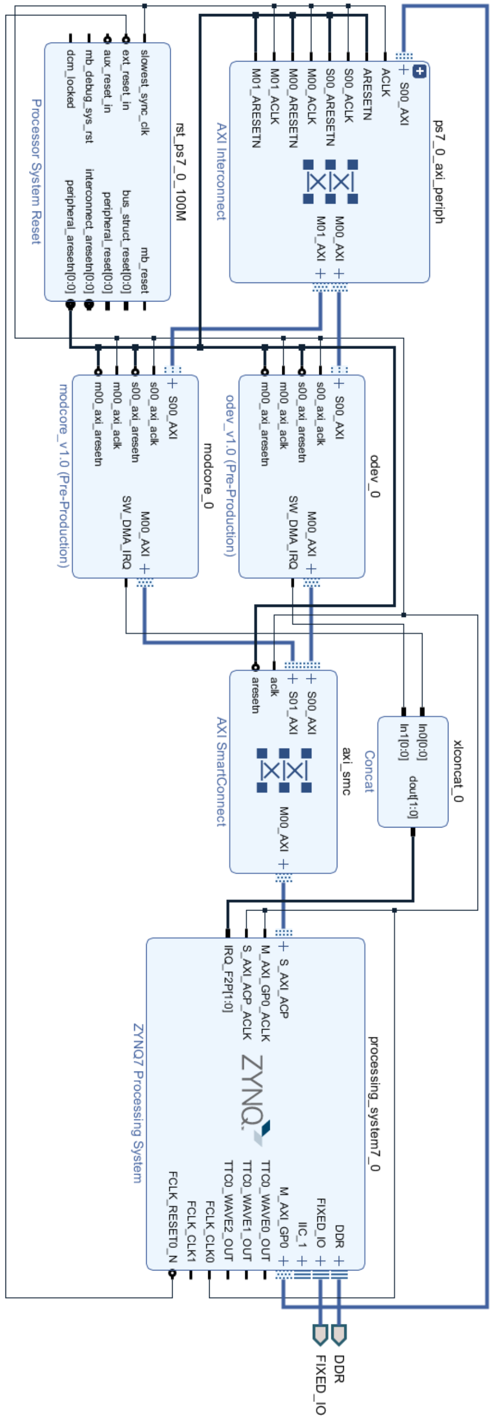


Figure 7-24: Block design in hardware (Modcore, odev) and software codesign

You can get the latest FPGA bitstream *(design_1_wrapper.bit)* that already contains the Modcore coprocessor hardware from the *tools* repository *(github.com/chungae9ri/tools)*. But if you installed the Vivado 2018.3, it is so easy to create the Vivado project for all FPGA hardware. The *genesis* directory has auto scripts to create a Vivado project for chapter 7. Double click the *genesis/create_genesis_project.bat.* This batch file launches Vivado and creates all FPGA IP modules and connect them. This will launch Vivado and runs the TCL scripts to build the final block design as in figure 7-24.

Look at the block design in figure 7-24. It has the odev_0 and modcore_0 block which are our custom FPGA peripheral hardwares. Those are connected to the PS Cortex ARM processor through S_AXI_ACP port. ACP port is a 'Accelerator Coherency Port' which is connected to the Snoop Control Unit (see figure 2-22 Zynq7000 subblock diagram). By connecting the peripheral hardware to the SCU, the DMA cache coherency between PS ARM processor and peripheral DMA is supported by hardware (SCU).

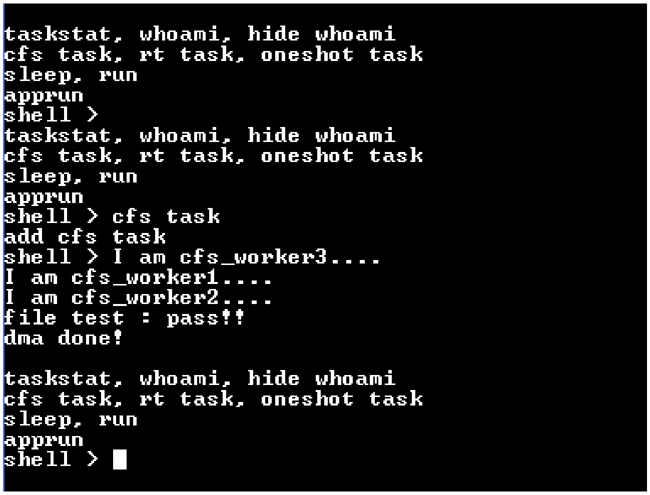
Notice that the interrupt signal from each hardware module is gathered in *xlconcat* module and go to IRQ_F2P in the PS subsystem. The IRQ_F2P is the port for the interrupts from PL subsystem to PS subsystem.

All these steps are automated by script files. To build the bitstream, just click *Generate Bitstream* menu in the Vivado.

Then, let's build a BOOT.BIN with the latest FPGA bitstream*,* ramdisk.img*,* kernel.elf*,* zynq_fsbl_hook.elf as we did before*.* Then, copy the BOOT.BIN into SD card and try boot the Zynq board. After bootup, run a 'cfs task' command to create CFS tasks. If you see the "dma done!" message, then the Modcore coprocessor successfully completes its 256's complement jobs for the 64KB data in 0x2000_0000. You can check the result in 0x3000_0000 with GDB memory check command. Since the source data has a predefined pattern 0x00 0x01 0x02 ... 0xFF 0x00..., the data copied into destination must be 0x00 0xFF 0xFE 0xFD ... In addition to Modcore coprocessor test, the *cfs_worker2* runs previous SLFS test. When you run the test, you also should see the result of SLFS file test message. The result of this test is demonstrated in figure 7-25.

Next test is the outstream device. Comment out the CFS worker3 for disabling the modcore task and enabling the CFS worker4 for enabling the outstream task. Build SLOS and boot up SLOS. Run 'cfs task' command in the shell. This will start the worker4 and you can see the text scrolls up to itab entry 575. This is becuase the data consumer isn't enabled and all buffers in the block are full now.

Now run a 'start cs' command in the shell. This will start the data consumer block and the data flows through the datapath again. After hitting the 4096th entry, the data round back to the first location of the memory and runs forever until there is a sequence error happens. If sequence error occurs, all blocks stop and sequence error interrupt will be fired to the PS subsystem. So, the software running in the PS subsystem can recognize the error in the hardware and can handle it properly.



```
taskstat, whoami, hide whoami
cfs task, rt task, oneshot task
sleep, run
apprun
shell >
taskstat, whoami, hide whoami
cfs task, rt task, oneshot task
sleep, run
apprun
shell > cfs task
add cfs task
shell > I am cfs_worker3....
I am cfs_worker1....
I am cfs_worker2....
file test : pass!!
dma done!

taskstat, whoami, hide whoami
cfs task, rt task, oneshot task
sleep, run
apprun
shell > █
```

Figure 7-25: Test result for Modcore coprocessor

7.7 Summary

In this chapter 7, we demonstrate a simple practice of hardware-software codesign. We partitioned the job (performing 256's complement on each byte in memory) into software part and hardware part. Software part performs programming of hardware and handles the interrupt. Hardware part performs DMA work and the logical work (256's complement). We also defined the interface between hardware and software. The AXI interface is used for the physical interface bwteen them. The register R/W and interrupt is a logical layer of this hardware-software interface. Register R/W is used for DMA device driver (software) to program the Modcore (hardware), and the interrupt is used for the Modcore to notify an event to the DMA device driver.

After defining these hardware-software codesign stuff, we create a new FPGA bitstream with a Modcore coprocessor and an SLOS with a DMA device driver. The DMA device driver added a new interrupt into the GIC controller. Modcore coprocessor also can be easily programmed by using a VHDL. The final version of *kernel.elf* and FPGA bitstream shows our hardware-software codesign works very well.

Reference

[1] Jasinski, Rcardo. *Effective Coding with VHDL, PRINCIPLES AND BEST PRACTICE*, The MIT press, 2016

[2] ARM, *AMBA AXI and ACE Protocol Specification*

[3] Xilinx, *AXI Reference Guide, UG761*

[4] Xilinx, *Vivado Design Suite User Guide, Creating and Packaging Custom IP, UG1118*

[5] ARM, *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*

[6] ARM Cortex-A Series Programmer's Guide

8 Symmetric Multi-Processor

8.1 Introduction

The Zynq 7000 chipset has Cortex-A9 dual core for its PS subsystem. The secondary CPU core is in its *WFI (Wait For Interrupt)* while the bootup process is undergoing. Until this chpater, SLOS doesn't do anything for the CPU 1. All SLOS tasks are running in the CPU 0. In this chpater, we will activate the secondary CPU and change SLOS to support the *SMP (Symmetric Multicore Processor).*

SMP OS, as the name implies, doesn't have any CPU preferences in running its tasks. SMP SLOS runs the same code in both CPU 0 and CPU 1. Each CPU processor has its own variable storage to maintain its own resources such as runq, waitq, timer framework and so on. Nonetheless, the logical implementations for all these resources are exactly identical and the same programming codes are shared between the CPUs. So, the same SLOS implementations are just repeated in the CPU 1 for the SMP SLOS.

Since we want to keep SLOS as simple as possbile, we don't do complex features in this chapter. We will boot up the secondary CPU, demonstrate an Inter-Processor Communications by using an *SGI (Software Generated Interrupt)*. We will also migrate the outstream task which was implemented in the chapter 7 into the secondary CPU. This is good for task load balancing because the CPU 0 has shell task, RT tasks and the CPU 1 has outstream task.

There is another type of OS to support multiple cores. The *AMP (Asymmetric Multicore Processor)* is running different OS in a different processor. For example, high level OS such as embedded Linux is running on the CPU 0 and application specific OS such as RTOS or just bare metal application is running on the CPU 1. By running application specific features are implemented and running a dedicated CPU processor, AMP OS could be a better choice for some embedded system. There are many chipset vendor specific articles or wiki to support AMP implementation in their chipsets. But we don't cover the AMP for SLOS.

Figure 8-1 shows new blocks added in chapter 8. Actually, they are not new block, rather they are just difference instance of the same block; each block has its own resource storage in the memory. But each gray-ed block runs only in its associated CPU and the logical implementation is identical. Those blocks are interrupt blocks, exception blocks, memory management blocks, timer framework blocks and CFS scheduler blocks. The same page tables are used in both CPUs, which means both CPUs sees the same memory space. This is good and reasonable for SMP SLOS. But the task specific stacks should be allocated for each CPUs. We will cover the new memory map in the next chapter. Since this is an SMP SLOS, the rest module blocks can be run in any CPUs. This makes it possible to share the hardware resources such as uart HW between CPU 0 and CPU 1. CPU 0 and CPU 1 both can print its message through the uart terminal. There is a synchronization issue between them and we may see text message jumbled between CPUs and someitmes unstable operation of SLOS, but we don't care the synchronization issue in this case. We take care of the CPU synchronization issue only for the CPU's mailbox which we will implement later in this chapter. So, we can run any tasks in any CPUs except the grayed blocks.

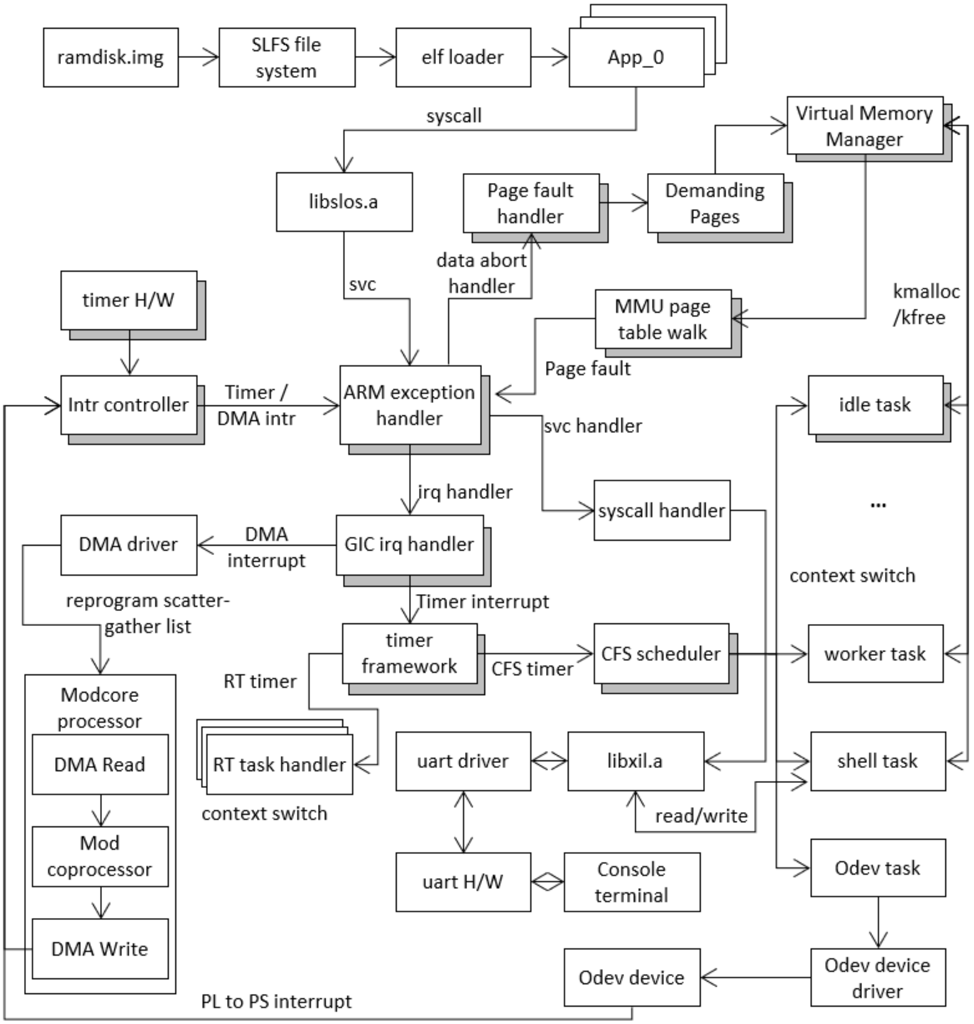


Figure 8-1: Top view of SMP SLOS. Gray block is duplicated through different CPUs.

The SLOS source tree can be checked out by either using tag or checking out with branch name. You can use *'git checkout SLOS_CH8'* or *'git checkout v2.0'*. The *meta-slos* is also tagged with v2.0 and you can use the bitbake to build the SLOS for chapter 8. For bitbake build, refer to the Appendix A. For SMP SLOS implmentations, SLOS source files has many changes across the source file tree. After checking out chapter 8 branch, let's first look at the new files.

1) *kernel/arm/smp-vector.S*
   This is an exception vector handler for the secondary CPU. The implementation is the same with the CPU 0 but the exception vector address, stack addresses are different. The sceondary CPU supports only reset vector, data abort handler for demanding page and IRQ handler.

2) *kernel/arm/smp-reset.S*
   This file has the implementation to restart the secondary CPU. It is very simple assembler code to load the reset vector address and jump to that address. The jump address that is the start address of secondary stage bootloader (*ssbl)* is stored while the CPU 0 starts up.

3) *kernel/core/percpu.c*
   This file declares the per CPU variables stored in the CPU specific locations.

4) *kernel/core/sgi.c, kernel/inc/sgi.h*
   This file has the SGI interrupt handler implmentations and its header file.

5) *kernel/core/mailbox.c, kernel/inc/mailbox.h*
   This file has a mailbox implementation for the inter processor communication. Mailbox stores letters coming from the other processor. Mailbox is implemented on top of the SGI interrupt.

6) *kernel/inc/percpu.h*
   This file has percpu declarations that can be included from different source files.

7) *kernel/in/percpudef.h*
   This file has the implementations of the macro *DEFINE_PER_CPU*. This macro definition is used in the percpu.c to declare CPU specific variable storage.

In addition to these new files, the SMP implementation comes from timer.c, ktimer.c, runq.c, waitq.c, sched.c, task.c, gic.c and main.c. And from this chapter, SLOS starts to provide the *TASK_WAITING* state properly by implementing the *msleep()* and *usleep()* functions. This fully supported task state will be demonstrated in the later of this chpater.

8.2 SMP SLOS Memory Map

As mentioned in chapter 8.1, SMP SLOS assigns CPU specific resource storage. Secondary CPU will have a storage for its own exception handler, virtual memory management, timer framework and CFS scheduler. The gray box in the figure 8-1 shows another instance of these block for the secondary CPU. Other blocks can be run in either CPU 0 or CPU 1 but not in both CPUs simultaneously. At the end of this chapter, we will move the outstream device driver task of PL subsystem hardware (Odev hardware device) into the secondary CPU.

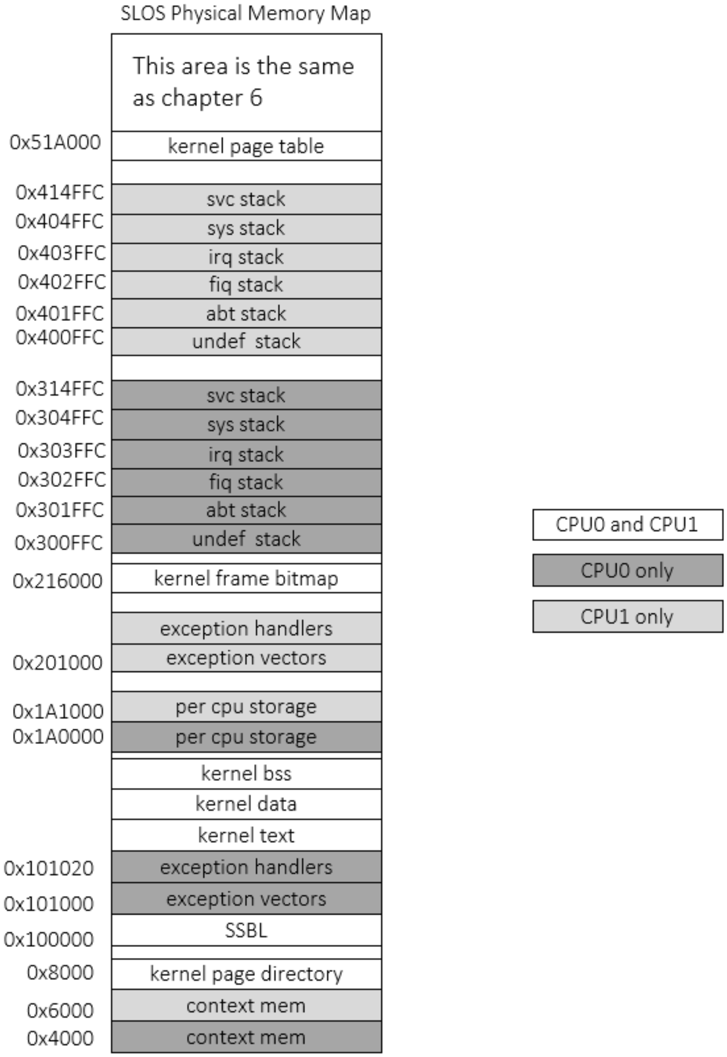


Figure 8-2: Memory map for SMP SLOS

CPU 1 exception vectors starts at 0x201000 and stacks are placed between 0x400000 ~ 0x415000. Per CPU storages are placed in 0x1A0000 for CPU 0 and 0x1A1000 for CPU 1. But all others are shared between CPU 0 and CPU 1. The SSBL code, kernel text, data, bss and page tables are all commonly used for CPU 0 and CPU 1. This means the hardware resources such as external memory, UART peripheral are all common resources and accessed by both CPUs. We don't care the synchronizations for these accesses, which means the uart message could be jumbled by each CPU. But later, when we cover the mailbox for IPC, we have to take care of the synchronization of accesses from both CPUs.

To reflect this new memory map, the linker script is also changed. Following in the new linker script used in chapter 8.

```
SECTIONS
{
    . = 0x00100000;
    .ssbl : {
        *(SSBL);
        *(PGT_INIT);
    }

    . = 0xC0101000;
    .text : AT(ADDR(.text) - 0xC0000000) {
        *(EXCEPTIONS);
        *(.text);
    }

    .data : AT(ADDR(.data) - 0xC0000000) {
        *(.data);
        *(.mailbox);
    }

    .bss : AT(ADDR(.bss) - 0xC0000000) {
        *(.bss);
    }

    . = 0xC01A0000;
    .data.percpu : AT(ADDR(.data.percpu) - 0xC0000000) {
        *(.data.percpu);
        }

        . = 0xC0201000;
        .secexceptions : AT(ADDR(.secexceptions) - 0xC0000000) {
            *(SECONDARY_EXCEPTIONS);
        }

        . = KERNEL_HEAP_START;
        .kheap : AT(ADDR(.kheap) - 0xC0000000) {
            __kernel_heap_start__ = .;
            *(.kheap)
            . = __kernel_heap_start__ + KERNEL_HEAP_SIZE;
            __kernel_heap_end__ = .;
        }
    }
```

Notice that the *'.data.percpu'* section and *'.secexceptions'* section are newly added. All others remain the same. Per CPU 0 section is placed at 0x1A0000 and per CPU 1 storage location is determined by the offset 0x1000 from the same variable of CPU 0. CPU 1 exception handler is placed at the address 0x2010000 as in the memory map.

8.3 SMP bootup

SMP bootup process starts from the software running in the CPU0. After finishing up its booting process, the software, operating system, running in the CPU0 is responsible to reset the CPU1 and put the right jump address for the CPU1 reset handler. Then, the secondary_start_kernel for CPU1 starts and operating system runs on its own. Before resetting CPU1, CPU1 is running in WFI mode. In Zynq7000, WFI in the CPU1 is programmed by the BootRom. Figure 8-? summarizes the SMP booting process.

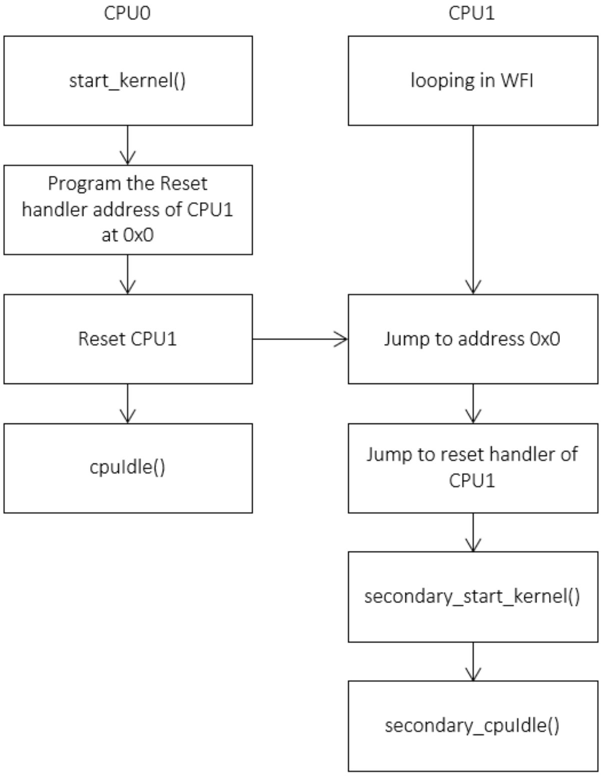


Figure 8-2: CPU0 and CPU1 bootup sequence

CPU0 and CPU1 run symmetrically and share the same operating system running in both CPUs. But while booting up, the CPU0 is a primary CPU and responsible for CPU1 booting. CPU1 is a secondary CPU and dependent on the initiation from CPU0.
As one operating system code is running in both CPU0 and CPU1, sometimes the operating system needs to know which processor it currently running on. Operating system can figure out current running processor by reading the *MPIDR (Multi-Processor ID Register)* register. Cortex-A9 technical reference manual defines this register.

31	30	29 12	11 8	7 2	1 0
1	U	SBZ	Cluster ID	SBZ	CPU ID

Figure 8-3: MPIDR bit assignment

1) bit[31]
   Indicates the register uses the new multiprocessor format. This is always 1.
2) bit[30]

U bit. Multiplrocessing extensions.
0: Processor is part of an MPCore cluster
1: Processor is a uniprocessor

3) bit[29:12]
SBZ
4) bit[11:8]
Cluster ID. This defines a Cortex-A9 MPCore processor in a system with more than one Cortex-A9 MPCore processor present. SBZ for uniprocessor system.
5) bit[7:2]
SBZ
6) bit[1:0]
Indicates CPU ID in the Cortex-A9 MPCore configuration.
0 for CPU0, 1 for CPU1, 2 for CPU2, 3 for CPU3. Always 0 for uniprocessor.

Reading this register can be done as below.
MRC p15, 0, <Rd>, c0, c0, 5;

After reading and bit masking for bit[1:0], software can figure out what CPU it is currently running on. SLOS also accesses this register in several places.

8.3.1 Cortex-A9 CPUs in Zynq7000

As in figure 8-4, Zynq7000 chipset has exactly the same two CPUs. They have the same Cortex-A9, MMU, I / D L1 caches, FPU and NEON Engines. These blocks are banked and when access to the register of a block of a specific CPU, even though the register address is the same, it alwasys points to the register of the block associated with that CPU. This means, each block of a CPU should be initialized when each CPU starts up.
Followings are the blocks needed to be initialized when each CPU starts up.

1) MMU
Page table needs to be set up for each CPU
2) GIC distributor and CPUx interface
GIC banked registers needs to be set up for each CPU
3) D cache invalidate
D cache invalidate needs to be done before the secondary CPU starts

SCU unit is initialized only when CPU1 starts. The CPU interfaces and some banked register in GIC distributor are also needed to be handled in each CPU. When operating system needs to do different implementations according to CPUs, the *MPIDR* register is used to figure out the current CPUID.

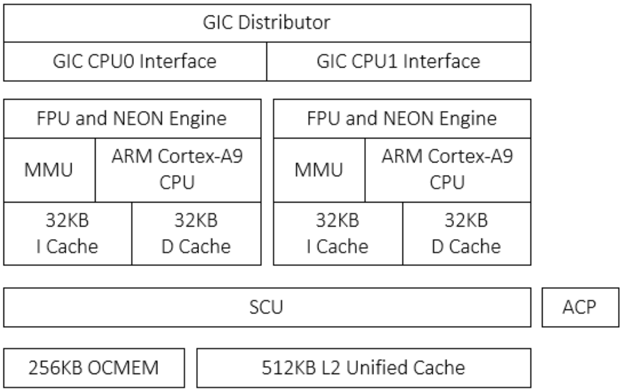


Figure 8-4: Cortex-A9 CPUs in Zynq7000

8.3.2 Primary CPU Initiatives

At the end of start_kernel() in the primary CPU, SLOS does two more things to bring up the second CPU.

1) Program (copy) the jump code into address 0x0
2) Reset the secondary CPU1

Programming the jump code into the address 0x0 is composed of below 3 ARM instructions (*kernel/arm/smp-reset.S*).

```
 1  .arm
 2
 3  .global secondary_reset
 4  secondary_reset:
 5              ldr     r0,   secondary_reset_jump
 6              bx      r0
 7  .global secondary_reset_jump
 8  secondary_reset_jump:
 9              /* cpu1 reset handler addr*/
10              .word     0
11  .global secondary_reset_jump_end
12  secondary_reset_jump_end:
13  secondary_reset_end:
```

It first loads the jump address in line 5. Then, it makes the PC to jump to that address in line 6. Line 10 reserves 4 bytes for saving the jump address.

These 3 bytes ARM ISA are loaded into address 0x0 as below (*kernel/core/main.c)*. The *start_cpu1()* is called at the end of start_kernel(), just before cpuidle().

```
void start_cpu1(void)
{
        uint32_t i, A9_rst_ctrl;

        A9_rst_ctrl = *(volatile uint32_t *)(A9_CPU_RST_CTRL);
        A9_rst_ctrl |= (A9_RST1_MASK | A9_CLKSTOP1_MASK);
        *(volatile uint32_t *)(A9_CPU_RST_CTRL) = A9_rst_ctrl;

        /*load [0x8] to r0*/
        *(volatile uint32_t *) (0x0) = *(uint32_t *)(secondary_reset);
        /* bx to [r0] */
        *(volatile uint32_t *) (0x4) = *(uint32_t *)(secondary_reset + 4);
        *(volatile uint32_t *) (0x8) = KERNEL_CODE_BASE;

        /* flush cache of cpu 0 */
        flush_ent_dcache();

        while (i < 1000)
                i++;

        // release cpu1 reset
        A9_rst_ctrl &= ~(A9_RST1_MASK);
        *(volatile uint32_t *)(A9_CPU_RST_CTRL) = A9_rst_ctrl;
        // release cpu1 clock
        A9_rst_ctrl &= ~(A9_CLKSTOP1_MASK);
        *(volatile uint32_t *)(A9_CPU_RST_CTRL) = A9_rst_ctrl;
}
```

The start_cpu1() function stops cpu1 processor by clearing the *A9_CPU_RST_CTRL* register. The address of this register is 0xF8F00244 from [1]. The bit assignments we are interested in are as follows.

1) bit[0]
   CPU0 reset, 1 for holding reset, 0 for releasing reset
2) bit[1]

CPU1 reset, 1 for holding reset, 0 for releasing reset

3) bit[4]
   CPU0 clock stop, 1 for stop clock, 0 for enable clock
4) bit[5]
   CPU1 clock stop, 1 for stop clock, 0 for enable clock

Line 5, 6, 7 does stop the clock and hold reset of CPU1. Then, line 9 ~ 13 copies the 3 ARM ISA into address 0x0. Take a notice on the *volatile* keyword when accessing the memory address. Before enabling the CPU1, we have to flush all the data cache into the point of coherency (PoC) to make sure the CPU1 has the same view of memory as CPU0. When CPU1 boots up, it enables SCU and invalidate its all caches to see the consistant memory view. Now, in line 21 ~ 26, the clock is enabled and reset signal is released and CPU1 starts to run from address 0x0. CPU0 continues its going and falls into the cpuidle().

When CPU1's PC is at address 0x0, it first sees an ARM ISA '*ldr, r0, [0x8]*'. We stored the start address of the CPU1 into the address 0x8. This address is loaded into the register r0, and next ISA is jump to this address.

8.3.3 Secondary CPU Startup

Basically, the startup code for CPU1 is almost the same as the handler of CPU0. This is because the processors are symmetrical; there is no HW differences between CPU0 and CPU1. As the startup of CPU0 starts from the memory initialization in the SSBL, CPU1 also starts from the same memory initialization. The memory initialization code of CPU0 is reused in CPU1 with minor modifications. The new implementations are as follows.

1) Enable SCU if current CPU is CPU1
2) Set SMP bit in ACTLR
3) Enable Cache/TLB maintenance broadcast
4) Jump to the different reset handler for each CPU

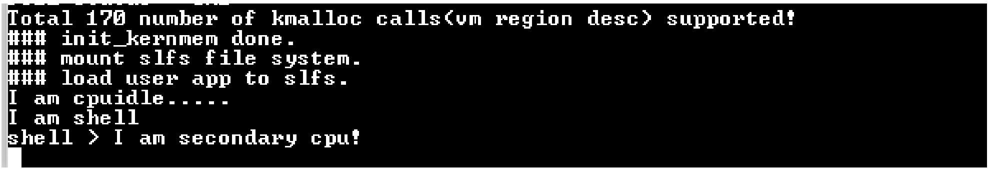
CPU0 and CPU1 shares the same memory map, TTBR and page tables. As stated, they share the same memory data and operating system code. The ssbl code in the init-mm.S runs two times for CPU0 and CPU1, but the *init_pgt()* in the mm.c file is run once only in the CPU0 bootup. The page table is shared between CPUs, it doesn't need to re-initialize the page tables two times. Refer to the *init-mm.S* file for defailed implementation for these.

The reset handler for CPU1 is exactly same except setting up the *VBAR* value. For CPU0, the exception base address is 0xC0101000 and for CPU1, this address is set as 0xC0201000. The reset handler sets the stack addresses for each processor modes then jumps to the secondary_start_kernel(). secondary_start_kernel() proceeds its initializations for CPU1 similarly with start_kernel() for CPU0.

Before adding more implementations into secondary_start_kernel(), let's add a print message to see whether the CPU1 booting is properly done. For now, secondary_start_kernel() is as simple as below.

```
int secondary_start_kernel(void)
{
        xil_printf("I am secondary cpu!\n");
        while(1);
        return 0;
}
```

If CPU1 properly boots up, the serial terminal should display a message like below screen shot.



```
Total 170 number of kmalloc calls(vm region desc) supported!
### init_kernmem done.
### mount slfs file system.
### load user app to slfs.
I am cpuidle.....
I am shell
shell > I am secondary cpu!
```

Figure 8-5: Boot up the secondary CPU

The uart device is accessed by both CPU0 and CPU1 simultaneously and the text message could be jumbled, but it is OK. While CPU1 keeps spinning in the infinite while loop, the CPU0 works exactly as before; it runs shell task, idle task and all other tasks.

8.3.4 Debugging CPU 1 with XSDB

Xilinx JTAG debugger can be attached to the CPU 1. So far, we attached the XSDB only to the CPU 0, but we can attach it easily by designating the target core. After running the XSDB and connecting it to the JTAG, run *'target 3'* at this time. Then you can see the '*' mark in the CPU 1. You can still use the GDB for CPU 0 debugging as we did before, but I can't figure out how to use the GDB for CPU 1 debugging.

In the XSDB window, run 'bpadd 0x0' to break the CPU 1 when it is reset by the boot primary core, CPU 0. After adding the breakpoint, run 'rst' command from XSDB. This command will reset both CPU 0 and CPU 1. Then, after the CPU 0 restarts the CPU 1 in its start_kernel(), the breakpoint should be hit and the CPU 1 should be under JTAG debugging. We can run XSDB command listed in the chapter 3.8.2. For example, we can run 'rrd' command to see the CPU registers' values. Or we can run 'dis pc 10' command to see the disassemble ARM machine code from current PC to 10 more lines. 'stpi' will step forward each disassembled code. If you want to add a breakpoint to a specific function, first you can get the function address by running the 'arm-none-eabi-objdump -t kernel.elf' command and then add the breakpoint of the specific function's address.

This debugging looks awkward, but definitely much better than nothing. We can debug both CPUs simultaneously by using XSDB for CPU 1 and GDB for CPU 0. For example, when we implement an SGI interrupt from CPU 0 to CPU 1, we can debug the CPU 0 with GDB and at the same time, we can add a breakpoint to the SGI interrupt handler of CPU 1 and do step each line by using XSDB. Sweet!

8.4 Enabling SLOS in the Secondary CPU

Now, we can migrate the SLOS implementations into the secondary CPU. Since this is an SMP, all SLOS tasks can run in any CPU and SLOS supports it with per_cpu interrupt, timer framework and scheduler. These blocks run in both CPUs and has the same implementations with different variable by using *per_cpu* storage. Each CPU has its own MMU as in figure 8-4 but in SLOS SMP, they are configured to have the same memory map and the same page tables as described in section 8.3.3. This was done while booting up the CPU 1.

Bofore going further, let's mention on the bankings in the ARM.

1) Interrupt Banking
   ARM has an interrupt banking that PPIs (Private Peripheral Interrupt) and SGIs (Software Generated Interrupt) can have multiple interrupts with the same interrupt ID. These interrupts are identified uniquely by the combination of its interrupt ID and its associated CPU interface.

2) Register Banking
   Register banking refers to implementing multiple copies of a register at the same address. This provides separate copies for each processor of registers in a multiprocessor implementation, or provides separate Secure and Non-secure copies of some registers in a GIC that implements the Security Extensions

In this chapter, we will use interrupt banking for SGI and register banking for private timer registers for the PPI. We will first port the current implementations of interrupt, timer framework and CFS scheduler into CPU 1.

8.4.1 Per CPU Variables

After booting up the second CPU successfully, we can transplant the implementations of the first CPU into the second CPU. The slos implementations for CPU 1 are almost the same as the CPU 0. The same code can be run in both CPU 0 and CPU 1 by associating some of the global variables with the specific CPU. The memory locations for the global variables of the CPU 0 are first assigned and then the memory locations of the CPU 1 variables are defined by the pre-defined offset from the corresponding CPU 0 variables. Because we are going to implment SMP slos, there is no difference between the slos codes running on the CPU 0 and CPU 1. Let's define the memory location for the variables of CPU 0 first.

percpu.h, percpudef.h and *percpu.c* files stores the newly added definitions of the per CPU variables. The *percpudef.h* has the macro definitions to dereference the per CPU variables. It looks like below.

```
 1    extern int smp_processor_id(void);
 2
 3    #define NR_CPUS                                   2
 4    extern unsigned long __per_cpu_offset[NR_CPUS];
 5
 6    #define per_cpu_offset(x) (__per_cpu_offset[x])
 7
 8    /* Separate out the type, so (int[3], foo) works. */
 9    #define DEFINE_PER_CPU(type, name) \
10            __attribute__((__section__(".data.percpu"))) __typeof__(type) per_cpu_##name
11
12    /* var is in discarded region: offset to particular copy we want */
13    #define RELOC_HIDE(ptr, off)                      \
14          ( { unsigned long __ptr;                    \
15            __ptr = (unsigned long)ptr;               \
16            (typeof(ptr))(__ptr + off); })
17
18    #define per_cpu(var, cpu) (*RELOC_HIDE(&per_cpu_##var, __per_cpu_offset[cpu]))
19    #define __get_cpu_var(var) per_cpu(var, smp_processor_id())
20    //
21    #define RELOC_ADDR(ptr, off)                      \
22          ( { unsigned long __ptr;                    \
23            __ptr = (unsigned long)ptr;               \
24            (unsigned long)(__ptr + off); })
25
26    #define per_cpu_addr(var, cpu) (RELOC_ADDR(&per_cpu_##var, __per_cpu_offset[cpu]))
27    #define __get_cpu_var_addr(var) per_cpu_addr(var, smp_processor_id())
```

Line 4 defines the offset values per CPU variable allocations and CPU 0 has offset 0 and CPU 1 has offset 0x1000 (4KB, 1 small page) from CPU 0.

Line 9 and 10 define a macro used to declare a CPU 0 variable with the section attribute. The *DEFINE_PER_CPU(type, name)* macro will put the CPU 0 variables into the *.data.percpu* section. The memory locations of these CPU 0 variables are assigned in the linker script. Following *.data.percpu* section comes after the *.bss* section in the linker script.

```
.data.percpu : AT(ADDR(.data.percpu) - 0xC0000000) {
        *(.data.percpu);
}
```

The memory address of these variable can be found by running *arm-none-eabi-objdump* command with *-t* option. As defined in line 9, 10, the per_cpu variables starts with *'per_cpu_',* such as per_cpu_idle_task.

The per_cpu variables for CPU 1 have offset from the per_cpu variables of CPU 0. Dereferencing these variables are defined in line 13 ~ 19. The *__get_cpu_var(var)* macro is expanded until it dereferences the pointer to the address of corresponding CPU's per_cpu variable. The content stored in that address could be just 4 byte data (uint32_t) or 4 byte address pointing to another place that is allocated by *kmalloc*.

When need to access the address, this macro can be used in the source code as below.

```
1   struct cfs_rq *this_runq = NULL;
2   __get_cpu_var(runq) = (struct cfs_rq *)kmalloc(sizeof(struct cfs_rq));
3   this_runq = __get_cpu_var(runq);
```

The __get_cpu_var(runq) will check the current CPU ID through the *MPIDR* register and returns the contents of the address. In upper case, line 2 fills out the per_cpu_runq memory location with the address returned from kmalloc(sizeof(struct cfs_rq)). In line 3, the this_runq variable is initialized by the address of per_cpu_runq and can be used in the coming code. If this code runs in the CPU 0, the value of this_runq is the content of the memory address per_cpu_runq for CPU 0 and if this code runs in the CPU 1, the value of this_runq is the content of the address (per_cpu_runq + 0x1000). Figure 8-6 illustrates this.

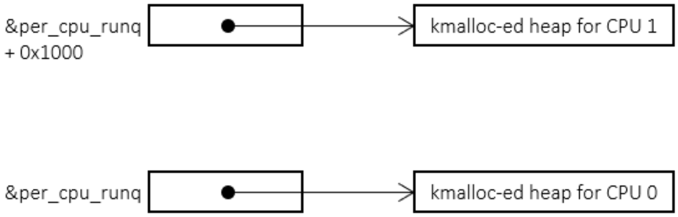


Figure 8-6: per_cpu variables and its content

From this, the same source code can be used for both CPU 0 and CPU 1. Variable this_runq is a local variable and initialized the kmalloc address for each CPU. The rest of implementation logic doesn't change. This is how SMP is working; same OS code / logic runs through all processors.

8.4.2 Enable Timer Interrupt in CPU 1

After CPU 1 boots up and each CPU has its own data with the per_cpu, next step is enabling the timer interrupt. We've made the secondary CPU jump to the *secondary_start_kernel()* function in section 8.2.3. The start routine was just printing a message to see whether the start-up routine is hit or not. Now, we need to add the CPU 1 initialization procedure into the previous simple start-up.

```
1    int secondary_start_kernel(void)
2    {
3            uint32_t ctrl;
4            xil_printf("I am secondary cpu!\n");
5            init_gic_secondary();
6            /* init timer */
7            *(volatile uint32_t *)(PRIV_TMR_LD) = 1000000;
8            gic_mask_interrupt(PRIV_TMR_INT_VEC);
9            *(volatile uint32_t *)(GIC_ICDICER0) = 0xDFFFFFFF;
10
11           /* enable timer */
12           ctrl = *(volatile uint32_t *)(PRIV_TMR_CTRL);
13           ctrl = ctrl | (PRIV_TMR_EN_MASK
14               | PRIV_TMR_AUTO_RE_MASK
15               | PRIV_TMR_IRQ_EN_MASK);
16
17           *(volatile uint32_t *)(PRIV_TMR_CTRL) = ctrl;
18
19           while (1);
20
21           return 0;
22   }
```

The newly added routine starts from line 5. First it initializes the GIC for the second CPU. The *init_gic_secondary()* does enabling the GIC distributor, CPU interface, and priority mask to forward all interrupt to the CPU 1. Then, line 6 ~ line 17 eanbles the CPU 1 private timer. The CPU private timer registers and its interrupt are all banked.

For now, there is no cpu_idle routine for CPU 1. So, the secondary_start_kernel() routine goes an infinite spin after initialize the gic and timer interrupt in line 19.

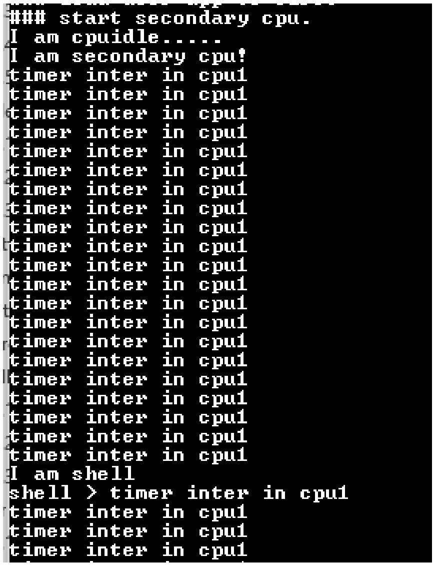
Next step is enabling an interrupt handler for CPU 1. This can be done by copying the interrupt handler from CPU 0. The IRQ interrupt handler is the same as CPU 0 except it is using 0x6000 for the context memory (CPU 0 is using 0x4000 for its context memory). There is no CPU ID specific routines here. The IRQ handler jumps the *gic_irq_handler()* whose implementation code is shared with CPU 0. The interrupt ACK register (GIC_ICCIAR) and end of interrupt register (GIC_ICCEOIR) in the gic_irq_handler body has the CPU ID for the information that shows the CPU requesting the current interrupt. But the CPU ID information from these registers is not used for now. The same gic_irq_handler function will be run in both CPUs. After it gets the interrupt ID, it calls exact the same specific interrupt service routines. In this case, the ISR must be an timer interrupt handler – *timer_irq().* The interrupt service routine functions were already registered while booting up the first CPU. The secondary CPU just reuses the prebuilt interrupt service routine vector.

To see whether the timer interrupt works well or not, let’s just a simple print message into the timer_irq.

```
id = smp_processor_id();
if (id == 1) {
      xil_printf("timer inter in cpu1\n");
      return 0;
}
```

This code checks the current processor id and if it is CPU 1, i.e., current timer interrupt is for CPU 1, then simply print a message and return. Because we don’t implement the timer frame work and any meaningful implementations, this is enough for now.

Let’s build SLOS and see if we can see the messages from CPU 1 timer interrupt handler. It should look like below figure 8-7.



```
### start secondary cpu.
I am cpuidle.....
I am secondary cpu!
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
I am shell
shell > timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
timer inter in cpu1
```

Figure 8-7: Checking timer interrupt for CPU 1

In figure 8-7, we can see the printing message and the message comes from either CPU 0 or CPU 1. The uart driver is shared between CPU 0 and CPU1. The shell task in the CPU 0 can still gets input from the terminal while CPU 1 timer interrupt handler prints message.

8.4.3 Memory Region for CPU 1

As defined in the figure 8-2, we have to define the exception vector address, stack address and *CONTEXT_MEM* address for CPU 1. These can be easily defined by applying offset to the corresponding addresses of CPU 0. SLOS defines the exception vector as 0x00201000, stack start address of CPU 1 as 0x0041_5000 and the context memory for context switching as 0x00006000. There are no specific rules for this, I just picked them from empty space. The new addresses are added to the *kernel/inc/mem_layout.h.*

The reset handler, irq handler of CPU 1 has exactly the same as CPU 0 but has the new addresses for CPU 1. Following is the irq_handler for CPU 1.

```
sec_irq_handler:
    ldr      r12, =SEC_CONTEXT_MEM
    sub      r12, #4
    stmfa    r12!, {r0-r11}
    sub      r0, lr, #4
    msr      cpsr_c, #MODE_SVC | I_BIT | F_BIT
```

```
        mov     r12, #0x602C
        stmfa   r12!, {sp, lr}
        str     r0, [r12, #4]
        bl      gic_irq_handler
        mrs     r0, CPSR
        bic     r1, r0, #I_BIT|F_BIT
        msr     cpsr_c, r1
        ldr     r12, =SEC_CONTEXT_MEM_END
        ldmfa   r12!, {r0-r11, sp, lr, pc}
```

As you can see in below CPU 1 irq handler implementation, it is exactly the same as CPU 0 except the address constant such as *SEC_CONTEXT_MEM.* Even it jumps the same *gic_irq_handler(),* which means CPU 0 and CPU 1 shares the interrupt service routines. Each subroutine runs based on the current CPU on which it is running, but there is only one interrupt service routine (*timer_irq)* in the interrupt vector table for the private timer interrupt. The *timer_irq()* conducts the timer_tree update, update sched ticks, and context switch on current CPU. So, the timer_irq is registered only once from the CPU 0 boot up sequence and it is reused in the CPU 1 timer interrupt.

To access CPU specific timer_tree and variables, the per_cpu location is used. To access timer_tree root, the local variable *this_root* is initialized with the CPU specific timer_tree root address as below.

*this_ptroot = (struct timer_root *)__get_cpu_var(ptroot);*

Then, the *this_root* variable can be used in the following implementation in both CPUs. This is possible in most implementations because the underlying logical implementations are identical to both CPUs in the SMP OS.

8.4.4 Enable Process Management in CPU 1

Now that we have separate variables, interrupt handler and private timer interrupt per CPU, it is right time to port the process management implementations of CPU 0 to CPU 1. Since the SLOS is an SMP, the implementations are identical and the same code is running on both CPUs. If current CPU needs to run different code, it can figure out current CPU ID by using *smp_processor_id()* as we did in the previous chpaters.

First, we have to define the variables per CPU. The file *kernel/core/percpu.c* contains the variables needed for each CPU.

```
DEFINE_PER_CPU(struct task_struct*, idle_task);
DEFINE_PER_CPU(struct task_struct*, current);
DEFINE_PER_CPU(struct task_struct*, last);
DEFINE_PER_CPU(struct task_struct*, first);
DEFINE_PER_CPU(uint32_t, task_created_num);
DEFINE_PER_CPU(struct cfs_rq*, runq);
DEFINE_PER_CPU(uint32_t, jiffies);
DEFINE_PER_CPU(struct timer_struct *, sched_timer);
DEFINE_PER_CPU(struct timer_root *, ptroot);
DEFINE_PER_CPU(struct clock_source_device*, csd);
DEFINE_PER_CPU(struct wait_queue*, wq);
DEFINE_PER_CPU(struct worker*, qworker);
```

These variables have its own storage locations per CPU. The addresses are defined as we did in chapter 8.3.1. Normally, these were global variables used in the single processor implementations, and now it is moved into the static storage which is determined in the compile and linkage time.

Each process management implementation can access the CPU specific variables as explained in chapter 8.3.1. CPU 1 has its own *idle_task, runq, timer framework, waitq and qworker* but the implementation is exactly the same as CPU 0. The detailed implementations aren't covered again.

8.5 Move PL Tasks to the CPU 1

In this chapter, we will do balance the work load between CPU 0 and CPU 1. We will move the outstream device task into the CPU 1. Since the shell task is still running in CPU 0, we need to have a method to notify the CPU 1 when shell task gets an order to create outstream task. For this, we will do initialization of GIC for CPU 1, enable one of *SGI* (*Software Generated Interrupt),* and add an interrupt service routine for this SGI.

8.5.1 GIC Initialization for CPU 1

Some of the GIC registers are banked, which means CPU 0 and CPU 1 has its own copy of the banked registers. If GIC Security extension is supported, some registers are banked between Secure mode and Non-Secure mode, but SLOS is always running in Secure mode and we don't need to take care of this. SGI and PPI interrupts are banked interrupt. The GIC initialization for the CPU 1 is done in the *init_gic_secondary()* function.

```
writel(0x0000FFFF, GIC_ICDICER0);
```

```
 2        writel(0xFFFFFFFF, GIC_ICDICER1);
 3        writel(0xFFFFFFFF, GIC_ICDICER2);
 4
 5        writel(0x1, GIC_ICDDCR);
 6
 7        writel(0xF8, GIC_ICCPMR);
 8
 9        writel(0x7, GIC_ICCICR);
10
11        writel(0xFFFF8000, GIC_ICDISER0);
```

The line 1 ~ 3 disables all interrupts except PPI interrupts (interrupt ID 16 ~ 31) by using *Interrupt Clear Enable Register (GIC_ICDICER).* Setting a bit in this register disables the corresponding interrupt. This register is also banked for each connected CPU and this doesn't affect the other CPU.

Line 5 enables the GIC Distributor. We did in configuring the GIC for CPU 0, but let's do it here again because this is an SMP OS.

Line 7 allows all 32 levels of interrupt priority to be forwarded to the CPU. Higher priority has lower priority field value. By default, all interrupts have the highest interrupt priority in the *interrupt priority regitser (GIC_ICDIPR).*

Then, line 11 enables all PPI interrupts by setting the corresponding bits in the *interrupt set enable register (GIC_ICDISER).*

After the init_gic_secondary(), GIC is initialized and enbles all PPI interrupts including the SGI interrupt #15. In next chapter, we will use this SGI #15 interrupt for the communication between CPU 0 and CPU 1.

Next change in the GIC implementation is from *gic_mask_interrupt().* This function does two operations before, which is 1) sets the interrupt target CPU interface and 2) sets the corresponding bits in the *GIC_ICDISER* to enable the interrupt forwarding to the CPU interface. Since SLOS has moved to SMP, we have to decide which CPU current interrupt is forwarded to. When the CPU 0 initializes the GIC distributor, it sets all interrupts to be forwarded to the CPU 0 only. By adding the current CPU ID into the *GIC_ICDIPTR* register in the gic_mask_interrupt(), the target of interrupt can be reprogrammed when enable the specific interrupt. The commented new implementation gic_mask_interrupt() looks as below. Notice that getting the current CPU id in line 2 and reprogram the target CPU id between line 4 ~ 11. But setting target CPU for SGI and PPI interrupts doesn't work because they are Ready-Only and changing target CPU for those interrupts is meaningless.

```
 1          /* get current cpuid */
```

```
cpuid = smp_processor_id();
/* reprogram GIC_ICDIPTR */
reg = GIC_ICDIPTR0 + (uint32_t)(vec / 4) * 4;
byte = (uint32_t)(vec % 4);
val = readl(reg);
/* clear current cpuid in the byte */
val = val & (0xFFFFFFFF & (0x00 << (byte * 8)));
/* set the cpuid in the byte */
val = val | ((0x1 << cpuid) << (byte * 8));
writel(val, reg);

/* banked register set-enable ICDISER0 */
reg = GIC_ICDISER0 + (uint32_t)(vec / 32) * 4;
bit = 1 << (vec & 0x1F);

/*
 * writing 1 enables intr
 * writing 0 has no effect
 */
writel(bit, reg);
```

8.5.2 Adding SGI Interrupt Handler

Many of registers for SGI interrupt are Read-Only, such as configuration register (*GIC_ICDICFR),* interrupt processor target register (*GIC_ICDIPTR).* These registers have a default reset value and don’t change during operation. It must be useless to forward CPU 0 private timer interrupt to CPU 1, so that configuration is determined at reset and is never changed.

Each CPU can interrupt itself, the other CPU, or both CPUs using a software generated interrupt (SGI). There are 16 software generated interrupts. An SGI is generated by writing the SGI interrupt number to the ICDSGIR register and specifying the target CPU(s). This write occurs via the CPU's own private bus. Each CPU has its own set of SGI registers to generate one or more of the 16 software generated interrupts. The interrupts are cleared by reading the ICCIAR (Interrupt Acknowledge) register or writing a 1 to the corresponding bits of the ICDICPR (Interrupt Clear-Pending) register [Zynq7000 TRM].

SGI interrupt handler is registered during the secondary_start_kerenl(). The interrupt handler registration looks like below.

```
1    void enable_sgi_irq(int vec, int (sgi_irq_handler)(void *arg))
2    {
3            gic_register_int_handler(vec, sgi_irq_handler, NULL);
4    }
```

In line 3, this function registers the SGI interrupt handler to the ISR vector table which was already used in the chapter 4 process management. We may register different SGI handlers per CPU but for this demonstration, we register only one SGI handler for CPU 1 only. The first paramger *vec* is set by the SGI interrupt number and used for the entry index of ISR vector table. We will use SGI #15 for the vec value. The second paramter is the SGI interrupt handler function pointer and it looks like below.

```
int sgi_irq(void *arg)
{
    struct sgi_data *pdat;
    pdat = (struct sgi_data *)arg;
       xil_printf("sgi intr %d from cpu: %d\n", pdat->num, pdat->cpuid);
    enqueue_workq(create_odev_task, NULL);

    return 0;
}
```

The *sgi_irq()* function is called from *gic_irq_handler()* as other interrupt but SGI interrupt has an CPU ID field in the *GIC_ICCIAR* register. We can figure out the source CPU by masking this field. In this demonstration, the sgi_irq() handler just does enqueue a work task function into the workq of CPU 1. As CPU 0 has its own *workq* task to handle the bottom half of the interrupt, CPU 1 also has its own *workq* named *"workq_worker:1"*. We delegate the bottom half of SGI interrupt. The bottom half of SGI #15 interrupt creates an outstream device task. In chapter 7.5, the outstream task runs in the CPU 0, but in this demonstration, the outstream task will run in CPU 1.

Creating the outstream task is conducted by the *forkyi()* function. The forkyi() function is exactly the same except the task stack base address is different on current running CPU. So, running the outstream task is nothing new.

Next, we need to add another shell command to test the SGI interrupt implementation. Adding an 'sgi' command to the shell task is as simple as adding a string comparison like other command. Then, that command sets the *SGIR* register as below.

```
uint32_t sgir = 0x0002000F;
```

```
2       *(volatile uint32_t *)(0xF8F01F00) = sgir;
```

Line 1 sets a value for SGIR register. It sets an SGI interrupt #15 and the target CPU 1. Detailed SGIR bit assignment is as below.

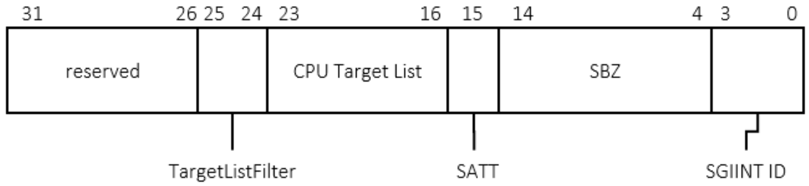


Figure 8-8: SGIR (SGI Interrupt Register) bit assignment

1) TargetListFilter
   2b00: send the interrupt to the CPU interfaces specified in the CPUTargetList field
   2b01: send the interrupt to all CPU interfaces except the CPU interface that requested the interrupt
   2b10: send the interrupt on only to the CPU interface that requested the interrupt
   2b11: reserved
   SLOS uses 0b00 for this field to use target list field.

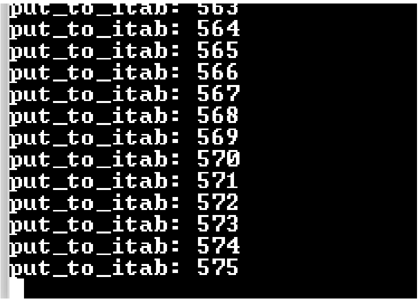
2) CPU Target List
   When TargetListFilter is 2b00, defines the CPU interfaces the Distributor must send the interrupt to. Each bit refers to the corresponding CPU interface.
   This field is set with 2b01 in our demonstration that SGI interrupt source is CPU 0 and target is CPU 1.

3) SATT
   Determines the condition for sending the SGI specified in the SGIINTID field to a specified CPU interfaces:
   0: only if the SGI is configured as Secure on that interface.
   1: only if the SGI is configured as Non-secure on that interface.
   SLOS set this value 0. SLOS always runs in Secure mode.

4) SGIINTID
   The Interrupt ID of the SGI to send to the specified CPU interfaces.
   SLOS uses SGI interrupt #15 and sets this value 0xF.

The right value to set the SGIR register should be 0x0002_000F and line 2 writes that value into the SGIR register.

Now we moved the outstream task to the CPU 1 and the shell task has a new command to generate an SGI interrupt to start the outstream task in the CPU 1. It's time to see whether it works or not. Build the SLOS BOOT.BIN and start it from the target board. After booting is completed, type 'sgi' command from the shell prompt. Then, the outstream task should start as below figure 8-9.



```
put_to_itab: 563
put_to_itab: 564
put_to_itab: 565
put_to_itab: 566
put_to_itab: 567
put_to_itab: 568
put_to_itab: 569
put_to_itab: 570
put_to_itab: 571
put_to_itab: 572
put_to_itab: 573
put_to_itab: 574
put_to_itab: 575
```

Figure 8-9: Outstream task output string

In figure 8-9, the outstream task is blocking because the itab table is full. Let's start the data consumer module to decrement the data. Run 'start cs' command in the shell. We can run this command from CPU 0 because CPU 0 and CPU 1 both have the same memory mapped IO and writing to the outstream register can be done from any CPUs. After running 'start cs' command from CPU 0, the data consumer module consumes the data and the outstream task runs again. If you want to stop the consumer module, run 'stop cs', then after a second, when the itab table is full again, the outstream task is waiting again.

Now, we have enabled the process management in the CPU 1, SGI interrupt between CPU 0 and CPU 1 and finally demonstrated this with outstream task.

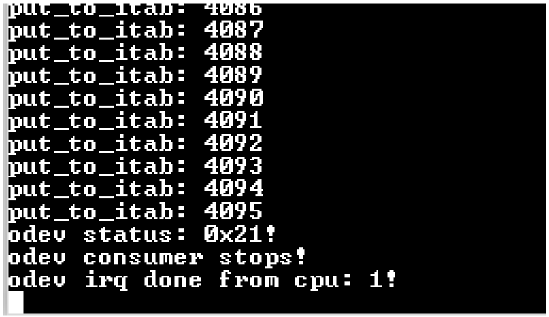
8.5.3 Adding Odev Interrupt Handler

We have started a worker task of outstream device in the previous chapter. Now we will move the interrupt handler of outstream device into the CPU 1. Because we changed the gic_mask_interrupt() to adapt the target processor in chapter 8.4.1, this is as easy as we need to call the gic_mask_interrupt() in the CPU 1. The outstream device interrupt number remains unchanged. This is done from init_odev() in the secondary_start_kernel(). Since all these functions are called from CPU 1, the gic_mask_interrupt() will automatically register the target CPU 1.

For testing, we need to generate a sequence error on purpose. We can do this easily by setting a wrong sequence number into the first frame when the outstream task wrap around the burst memory. Add following code into the outstream task, build and try to boot the target board.

```
/* make a sequence error */
if (i == O_STREAM_WRAP) {
    *(uint32_t *)psrc = 0xFFFFFFFF;
}
```

After all 4096 packet frames are sent to the DataConsumer module, then the outstream task wraps around to the first location of the burst memory for the 4097th frame. The outstream hardware module in the PL subsystem checks the sequence number and assert the interrupt signal to the PS. The legacy outstream interrupt handler still works but it is called from CPU 1 because the interrupt is forwarded to the CPU 1. Outstream interrupt handler is called as below figure 8-10.



```
put_to_itab: 4086
put_to_itab: 4087
put_to_itab: 4088
put_to_itab: 4089
put_to_itab: 4090
put_to_itab: 4091
put_to_itab: 4092
put_to_itab: 4093
put_to_itab: 4094
put_to_itab: 4095
odev status: 0x21!
odev consumer stops!
odev irq done from cpu: 1!
```

Figure 8-10: odev hardware interrupt forwarded to the CPU 1

8.6 Mailbox Implementation between CPU 0 and CPU 1

While porting the process management from CPU 0 to CPU 1, we built a good working communication channel between CPU 0 and CPU 1. It has SGI interrupt from CPU 0 shell task to CPU SGI interrupt handler and we used this channel to start the outstream task from CPU 1. Now, let's expand this communication channel to be more general channel. We will implement a mailbox for each CPU and use the CPU 1 mailbox to initiate the outstream task or to display the task statistics of CPU 1.

8.6.1 Mailbox Implementation

Mailbox is just a global variable defined as below.

```
    __attribute__((__section__(".mailbox")))
```

```
                              struct mailbox_struct mailbox_0;
      __attribute__((__section__(".mailbox")))
                              struct mailbox_struct mailbox_1;
```

A special section named '.mailbox' is created and statically linked in the compile time. The new linker script includes this section to the output section. Since the address of these two mailbox is statically defined, any CPU can access the mailbox simply by using the address of mailbox.

The *struct mailbox_struct* is defined as below.

```
enum letter_type {
    EMPTY = 0,
    TASK_STAT = 1,
    TASK_ODEV = 2,
};

enum letter_status {
    READ = 0,
    NOT_READ = 1,
};

struct mailbox_struct {
    enum letter_status status;
    enum letter_type letter;
};
```

Mailbox has two member variables. 1) *enum letter_status status* is to store the mailbox access status which is either READ or NOT_READ. READ status means the letter in the mailbox is read by the client, then is outdated and can be overwritten by a new letter. NOT_READ means the letter is still fresh and the postman should not overwrite this letter with a new one. 2) *enum letter_type letter* is to store the letter itself. For now, the content of the letter is limited to one of *EMPTY, TASK_STAT, TASK_ODEV*. *EMPTY* is an empty letter for initialization, *TASK_STAT* is for showing the CPU 1's task statistics, and *TASK_ODEV* is for triggering the start of outstream task from the CPU 1.

There are two functions to handle the mailbox: *push_mail()* and *pull_mail().*

1) *void push_mail(enum letter_type letter)*

This function places a letter into the other CPU's mailbox. Since the mailbox is a shared resource between CPUs and is a critical section where the accesses should be synchronized. Whenever it accesses the mailbox, the access tries to acquire the spinlock. After acquiring the spinlock, then the mailbox flag (*enum letter_status)* is checked. If the letter in the mailbox is read (mailbox flag down), this function places a new letter into the mailbox and update the mailbox status (mailbox flag up), then release the spinlock. If current letter isn't yet read, this function loops in checking the mailbox until the previous letter is read.

2) *enum letter_type pull_mail(void)*
   This function pulls the letter from current CPU's mailbox. This function first tries to get a spinlock flag, if the letter is already read, this function returns an empty letter and if there is a new letter, this function returns that letter to the caller. Before return, it should release the spinlock.

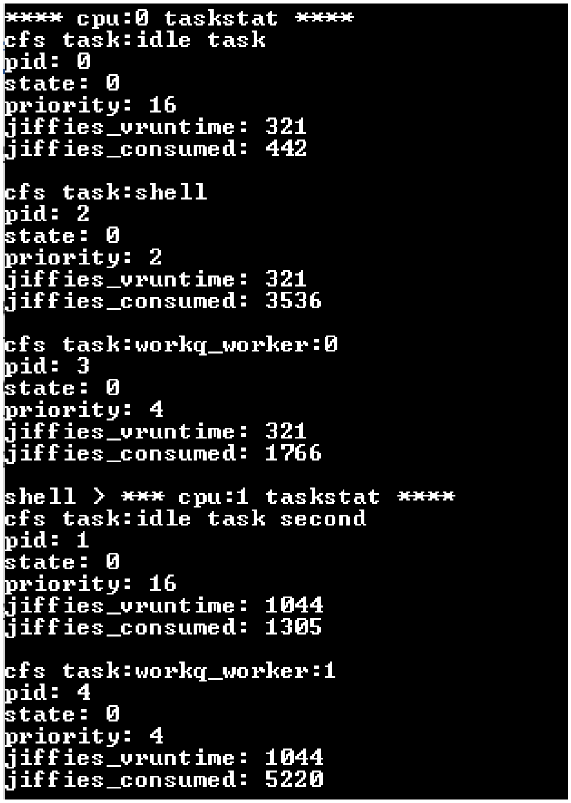
With simple mailbox and push / pull handler, we can add diverse communications between the CPUs. We may increase the mailbox capacity to store more letters. In this case, we have to add a flow control for each side not to overwrite the previous NOT_READ letter. We don't cover this for now, but will implement another type of letter message (TASK_STAT) in the next chapter.

8.6.2 Show TaskStat of CPU 1 by using Mailbox

Up until now, we don't have a way to show the statistics of the tasks running on the CPU 1. Since the shell task runs only in the CPU 0, we have to use a mailbox to send a letter that have the CPU 1 display its running tasks. In the previous chapter, we already defined the type of letters such as EMPTY, TASK_STAT, and TASK_ODEV. The letter type TASK_STAT is used for displaying the CPU 1 tasks' statistics. Below small SGI triggering code is added after the shell task's 'taskstat'command's *print_task_stat(NULL)* function.

```
enum letter_type letter = TASK_STAT;
push_mail(letter);
uint32_t sgir = 0x0002000F;
*(volatile uint32_t *)(0xF8F01F00) = sgir;
```

This code does two things, 1) push a TASK_STAT letter into the other CPU's mailbox and 2) generate a SGI interrupt. Then, the CPU 1's SGI interrupt handler will register the same function *print_task_stat()* to its workq worker task. This is what the top half of CPU 1 SGI interrupt handler does and returns. When the CFS scheduler of CPU 1 schedules the workq worker for next task, the workq worker picks and runs the next work registered to its workq which is the print_task_stat() function here. Sweet! Let's see the result of task statistics from both CPUs. It should look like below.



```
**** cpu:0 taskstat ****
cfs task:idle task
pid: 0
state: 0
priority: 16
jiffies_vruntime: 321
jiffies_consumed: 442

cfs task:shell
pid: 2
state: 0
priority: 2
jiffies_vruntime: 321
jiffies_consumed: 3536

cfs task:workq_worker:0
pid: 3
state: 0
priority: 4
jiffies_vruntime: 321
jiffies_consumed: 1766

shell > *** cpu:1 taskstat ****
cfs task:idle task second
pid: 1
state: 0
priority: 16
jiffies_vruntime: 1044
jiffies_consumed: 1305

cfs task:workq_worker:1
pid: 4
state: 0
priority: 4
jiffies_vruntime: 1044
jiffies_consumed: 5220
```

Figure 8-11: Statistics of the tasks running in CPU 0 and CPU 1

The cpu idle task in CPU 0 and CPU 1 has pid 0 and 1. CPU 0 has three tasks by default which are *cpuidle, shell,* and *workq_worker* tasks. CPU 1 has two tasks which are *secondary cpuidle,* and *workq_worker* tasks. The virtual runtime in each processor is fair among the tasks, shows CFS scheduler is still working well in both CPUs.

8.7 Putting it altogether

8.7.1 Adding Sleep Functions

From v2.0, SLOS starts to support sleep functions – *mslee(msec), usleep(usec).* Before v2.0, the *TASK_WAITING* state isn't fully supported. Let's look at the sleep functions first and how they can be used in the multiple cores.

The msleep(msec), usleep(usec) functions calculate the timer tick count corresponding to the msec or usec and then calls the *delay(ticks)* function. delay(ticks) function does follow things.

1) dequeue current task to the waitqueue
2) create oneshot timer by calling *create_oneshot_timer().* This oneshot timer will be fired in ticks count
3) calls *yield()* function to yield the processor to the current task's yield_task

When current task is dequeued to the waitqueue, its state is changed to the TASK_WAITING state, i.e., current task is blocking and doesn't consume CPU time. After yield() function is done, current task is switched to the yield_task.

After ticks time has passed, the oneshot timer fires an interrupt and the timer interrupt handler, *timer_irq(),* does follow things for the oneshot timer interrupt.

1) switch current task's context with oneshot timer task's context
2) enqueue oneshot timer's task into the runqueue

After oneshot timer interrupt handler returns, current task is changed to the blocking task and new task is in TASK_RUNNING, enqueued to the runqueue that the CFS scheduler schedules.

Currently, only CFS tasks except cpuidle task can use the sleep functions. There is at least one running task per CPU, cpuidle task can't be blocked and go to the waitqueue. Instead, cpuidle task conducts an important power saving routines as described in chapter 4.10. RT task calls yield() function voluntarily when it finishes its work. Using sleep() function in the RT task looks better and this may be changed to use sleep() function later.

In this chapter we will add the msleep() function to the outstream task which is always running after it starts. The outstream task's loop body is changed as below.

```
1   for (;;) {
2       if (!put_to_itab(O_STREAM_START + O_STREAM_STEP * i, O_STREAM_STEP)) {
3           xil_printf("put_to_itab: %d\n", i);
4           msleep(10);
5           i++;
6           i = i % O_STREAM_WRAP;
7       } else {
8           msleep(100);
9       }
10  }
```

As you see, the line 4 and line 8 replace the old while loop with msleep() function now.

Next change in v2.0 is that the workq worker task stays in blocking state until there is any work to run. In other words, workq worker task runs only when there is a work in its workq and falls back to TASK_WAITING state after finishing all its works. Workq worker task calls *dequeue_se_to_wq()* and *yield()* function at the end of its work. To wake up the workq worker, *enqueue_se_to_runq()* can be used. There is a *wakeup_workq_worker()* function to wake up current CPU's workq worker task.

8.7.2 Running Tasks in CPU 0 and CPU 1

Let's create more tasks running real time tasks in CPU 0 and outstream task in CPU 1. In this chapter, we will add two types of SGI interrupt handler – displaying CPU 1 task statistics and running the outstream task in CPU 1. The new SGI interrupt handler looks like below.

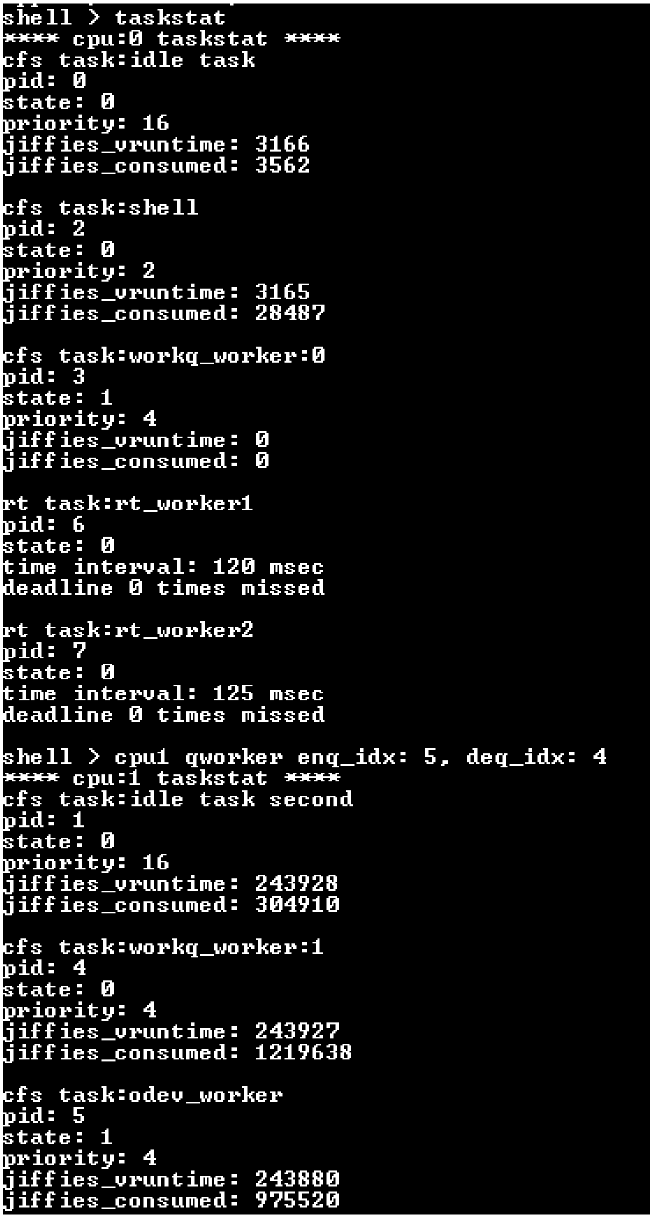
```
int sgi_irq(void *arg)
{
    enum letter_type letter = pull_mail();
    switch (letter) {
    case EMPTY:
                break;

    case TASK_STAT:
                enqueue_workq(print_task_stat, NULL);
                wakeup_workq_worker();
                break;

    case TASK_ODEV:
                enqueue_workq(create_odev_task, NULL);
                wakeup_workq_worker();
                break;

    default:
                break;
    }

    return 0;
}
```

SGI interrupt handler now handles all three types of the letters. This is top half of SGI interrupt handler and the bottom half is conducted by the workq_worker. Top half of each letter interrupt is composed of two steps. 1) Enqueue the corresponding task function to the workq worker queue. This can be done through *enqueue_workq()* function. enqueue_workq() function gets the task function as its first parameter and add it to the workqueue. 2) Then, the SGI interrupt handler wakes up the workq worker task by using *wakeup_workq_worker()* function. This function enqueues the current CPU's worker task into the runq for future scheduling. When CFS scheduler assign CPU time to the workq_worker, that task function will be picked from the workq and run by the workq worker.

The *pull_mail()* function in the line 3 pulls the letter from mailbox. Unlike push_mail() function which is blocked until the mailbox is free, pull_mail() function will return without waiting whether there is a valid letter or not. This is a right design because push_mail() is called from CFS task but the pull_mail() function is called from the interrupt context. Interrupt context should not be blocked (spinlock is OK but it should not spin for a long time) and sould return as fast as it can be.

From line 8 to 16, the SGI interrupt handler enqueues the associated task into the worker queue and wake up the worker task. For *TASK_STAT* message, it enqueues the *print_task_stat()* function to print out current CPU's task statistics. For *TASK_ODEV* message, it enqueues the *create_odev_task()* function to run the outstream task. The outstream task will switch between runq and waitq.

You can still use the *make* to build the v2.0, but let's use the bitbake to build the v2.0 SLOS. Bitbake build is described in the Appendix A. After running "bitbake world -vDDD", bitbake will automatically download toolchain, clone SLOS source files with tagging v2.0. If you want to build different tag, you can do that by editing the release tag number in the SLOS recipe. Let's create the BOOT.BIN and boots the target board. After boot up, from the CPU 0's shell task, run the "rt task" command to create the RT task in the CPU 0. Then, run "sgi" command to create outstream task in the CPU 1. Wait for a second until the ITAB is full. Then run "start cs" command to start the data consumer module in the outstream hardware. Following figure 8-12 is captured after running "stop cs".



```
shell > taskstat
**** cpu:0 taskstat ****
cfs task:idle task
pid: 0
state: 0
priority: 16
jiffies_vruntime: 3166
jiffies_consumed: 3562

cfs task:shell
pid: 2
state: 0
priority: 2
jiffies_vruntime: 3165
jiffies_consumed: 28487

cfs task:workq_worker:0
pid: 3
state: 1
priority: 4
jiffies_vruntime: 0
jiffies_consumed: 0

rt task:rt_worker1
pid: 6
state: 0
time interval: 120 msec
deadline 0 times missed

rt task:rt_worker2
pid: 7
state: 0
time interval: 125 msec
deadline 0 times missed

shell > cpu1 qworker enq_idx: 5, deq_idx: 4
**** cpu:1 taskstat ****
cfs task:idle task second
pid: 1
state: 0
priority: 16
jiffies_vruntime: 243928
jiffies_consumed: 304910

cfs task:workq_worker:1
pid: 4
state: 0
priority: 4
jiffies_vruntime: 243927
jiffies_consumed: 1219638

cfs task:odev_worker
pid: 5
state: 1
priority: 4
jiffies_vruntime: 243880
jiffies_consumed: 975520
```

Figure 8-12: Statistics of the tasks running in CPU 0 and CPU 1 (After adding Outstream task into the CPU 1)

Figure 8-12 shows the task information for CFS tasks in CPU 0 and CPU 1, RT tasks in CPU 0. Outstream task is one of the CFS task running in the CPU 1. Do they all look good to you? CFS tasks are all fairly assigned with the CPU time and realtime tasks doesn't miss the deadline and the outstream task in CPU 1 doesn't have a sequence error.

Nonetheless, the *virtual runtime* and *jiffies_consumed* in the CPU 1 is much faster than the CPU 0. I don't know why for now. Maybe the clock frequence could be different.
Check the task state of workq_worker:0 task and odev_worker task. They are in TASK_WAITING state. workq_worker:0 is always blocked and in TASK_WAITING state because there is no entry in its work queue. But the workq_worker:1 is in TASK_RUNNING state because when run the task_stat command, workq_worker:1 should be in the TASK_RUNNING to print CPU 1's task statistics. odev_worker's state is switching between TASK_RUNNING and TASK_WAITING. Because the sleep time is much longer than the rest of task's run time, mostly the outstream task is in TASK_WAITING state.

8.8 Summary

In this chapter, SLOS is changed to support the SMP. While SLOS boots up, it starts the secondary CPU at the end of the start_kernel(). To reset the secondary CPU, Zynq 7000 chipset has a A9_CPU_RST_CTRL register. How to boot up the secondary CPU was covered in chpater 8.2. The secondary CPU shares the memory map with CPU 0 and it sets up the same page tables as CPU 0. But there are still CPU specific global variables such as runq, timer red-black tree, waitq and so on. To support those CPU specific variables, SLOS implements the *per_cpu* variables. Each CPU can accesses its own variables by using the DEFINE_PERCPU macro and *__get_cpu_var()* function. Zynq7000 has a MPIDR (Multiprocess ID Register) to identify the CPU that current program is running on. The processor also has many registers banked for each CPU. The GIC is also needs to be configured to support both CPUs.

Since SMP SLOS runs the same code in both CPUs, any task can be run in any CPU. To demonstrate the SMP SLOS, the outstream task is moved to the CPU 1. Shell task is running in the CPU0 and the mailbox using the SGI interrupt is implemented to send a message between CPUs. By using the mailbox, the 'taskstat' and 'sgi' command from the shell task can propagate from CPU 0 to CPU 1. The SGI #16 is used for mailbox.

To support the task state such as TASK_RUNNING and TASK_WAITING, the sleep functions are reimplemented in this chapter. Sleep function makes the task go to the waitq and wake up after the programmed time pass. This also makes idle task (not cpuidle task) goes to waitq until it needs to be woken up. The workq worker task is mostly in the waitq and doesn't consume the CPU time until the wakeup_workq_worker() function runs.

Reference

[1] UG-585 Zynq7000 Technical Reference Manual

[2] ARM Cortex-A9 Technical Reference Manual

[3] ARM Cortex-A9 MPCore Technical Reference Manual

[4] Generic Interrupt Controller Architecture Specification

Appendix A – BitBake Build

A1. Introduction

BitBake user manual describes "BitBake is a *generic task execution engine* that allows shell and Python tasks to be run efficiently and in parallel while working within complex inter-task dependency constraints". Bitbake is originally developed and used by OpenEmbedded project and in 2004, it was splitted two distinct pieces (refer bitbake user manual document);

1) BitBake
   a generic task executor
2) OpenEmbedded (OE)
   a metadata set used by BitBake

BitBake is written in Python and runs the tasks based on the descriptions in the *class, conf* and *recipe.* OpenEmbedded project uses BitBake to construct complete Linux images. BitBake and OpenEmbedded are combined to form a reference build project named *Poky.*

On top of BitBake, OpenEmbedded project and Poky, Yocto is a superset of these to provide a complete set of Linux building (tools, cross toolchain, source fetching, patching, building, ramdisk building, etc.). For better understanding of the relationship among these projects, refer below figure A-1.

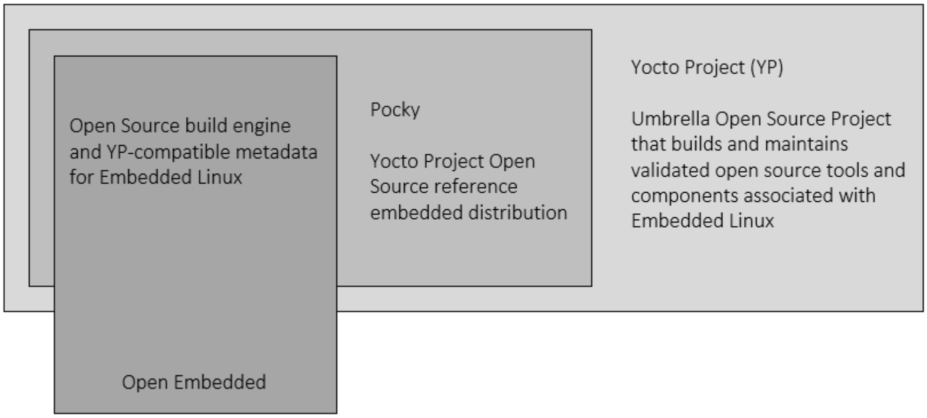

Yocto Project (YP)
Umbrella Open Source Project that builds and maintains validated open source tools and components associated with Embedded Linux
Pocky
Yocto Project Open Source reference embedded distribution
Open Source build engine and YP-compatible metadata for Embedded Linux
Open Embedded

Figure A-1: Relation among BitBake, OE, Poky and Yocto

As you can see in figure A1-1, Yocto is the superset of Poky and BitBake. Learning curve for Yocto is steep. It is not straightforward such *make menuconfig, make* command, but once the Yocto project runs, it performs many essential steps (source fetching, patching, packaging etc.) without the developer's involvement. See below figure A -2 for Yocto build.

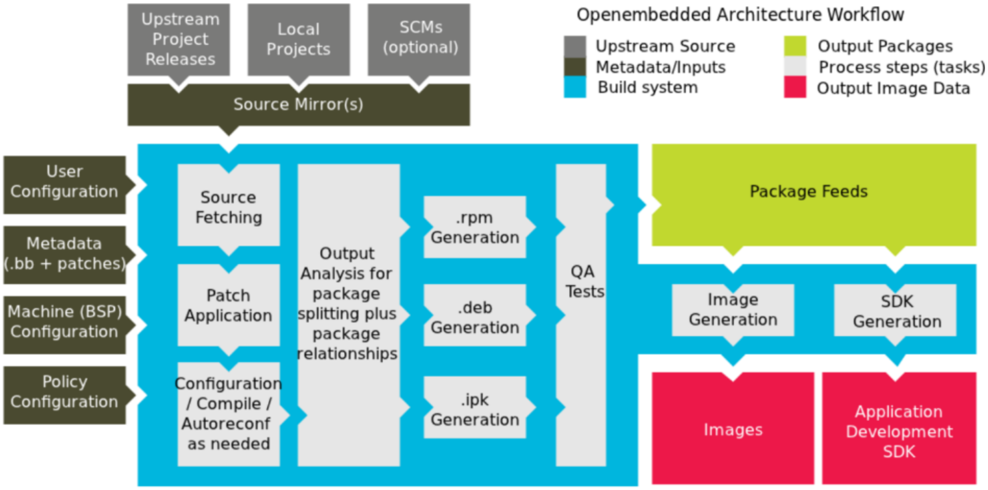


Figure A-2: Yocto Build Flow *(*ref: *Yocto Project Overview and Concept Manual)*

Different embedded Linux distribution has its own Yocto projects. Xilinx also moved Yocto build around 2016 and keeps maintaining its Yocto build projects such as *meta-xilinx.*

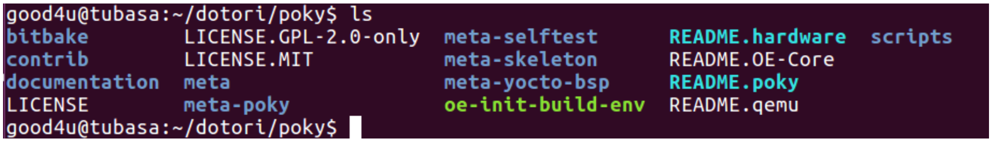
This appendix demonstrates how to use BitBake to build our custom SLOS. Building custom *meta* project will help us understanding the core process of BitBake and will be expanded to understand Yocto project as well.

A2. Meta-slos Build

Before creating custome layer for BitBake, let's see how current meta-slos recipes are used to build SLOS source files. Follow below steps to build SLOS. You don't need to install cross compiler toolchain for SLOS build if the BitBake and meta-slos are used.

1) Cloning Poky
   We are going to use BitBake in Poky framework. We can clone the BitBake repository and use it directly to build custom source tree such as SLOS, but using Poky gives us to use pre-defined features. In addition, these features are also used in Yocto build and it gives us some idea on how current Yocto build is working. Let's first clone the Poky by running below command in Ubuntu virtualbox.
   git clone git://git.yoctoproject.org/poky.git

After cloning done, you can see a directory named *poky*. Step into this directory. It looks like figure A - 3.



```
good4u@tubasa:~/dotori/poky$ ls
bitbake        LICENSE.GPL-2.0-only  meta-selftest     README.hardware  scripts
contrib        LICENSE.MIT           meta-skeleton     README.OE-Core
documentation  meta                  meta-yocto-bsp    README.poky
LICENSE        meta-poky             oe-init-build-env README.qemu
good4u@tubasa:~/dotori/poky$ 
```

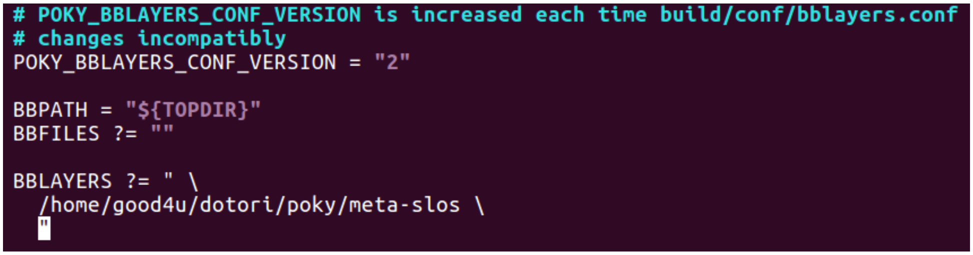
Figure A-3: Directories in Poky

The poky repository contains bitbake and prebuilt recipes starting with *meta-*. We will use BitBake as it is in Poky but will not use the recipes in the Poky. I will write our own recipes in *meta-slos.*

2) Building meta-slos
Before adding meta-slos recipes step by step, let's first experience how the meta-slos build works. Move out from current Poky directory and clone the meta-slos by running following command.
git clone https://github.com/chungae9ri/meta-slos

Then, move the cloned meta-slos directory to Poky directory by *mv meta-slos poky.* Next, go into Poky directory and run *source oe-init-build-env build-slos.* This command sets up SLOS build environment and create *build-slos* folder in the Poky and move current directory to that.

3) Now, you should be in *build-slos* directory. You can find that Poky already create *conf* directory and 3 files *(bblayers.conf, local.conf, templateconf.cfg)* for you. Open *conf/bblayers.conf* which contains which layers are built by BitBake. Go to *BBLAYERS* variable and replace all pre-added meta layers with the path to our meta-slos layer. It should be look like below.



```
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
  /home/good4u/dotori/poky/meta-slos \
  "
```

Figure A-4: bblayers.conf for building meta-slos

We can remove all meta layers except meta-slos to build SLOS. But we need to copy some files, let's not touch the meta layers.

4) Now, we are ready to build our custom SLOS. Close the bblayers.conf. Make sure that you are in build-slos directory. Poky build starts always in this location. Run *bitbake world -vDDD*. This will build everything in the meta layers. The option *-vDDD* is for displaying debugging messages. With this option, you can see all steps of BitBake task running. The BitBake uncompresses the toolchain (*gcc-arm-none-eabi-6-2017-q1-update-linux.tar.bz2*) and copies it to *${HOME}/bin/arm-2017q1*. Then, it clones the SLOS sources from github repository, and build it with GNU make.
   There is no need of user intervention; BitBake sets up the toolchain, pulls the SLOS source files, patches if needed, and builds the source files all at once.

5) Let's look into the output files. After build is done, a *tmp* directory is created. This directory contains all newly generated files such as downloaded sources, git directory, unpacked sources, built outputs, logs for each build step and so on. SLOS is built in *tmp/work/slos-1.0-r1/slos-1.0/git/out* directory.

A3. Steps to Create Meta-slos

In this chapter, the steps to create custom BitBake layer, meta-slos, are described. Following those steps can give you some sense on developing your own BitBake recipes and layer.

1) Clone the Poky and vanilla BitBake
   Cloning Poky was already explained in chapter A2. Clone BitBake by running *git clone https://github.com/openembedded/bitbake*. We will use only two files from the vanilla BitBake repository.

2) Create meta-slos directory in Poky. Then, create *classes, conf, recipes-slos* directories. You can run *mkdir classes conf recipes-slos* command for this. These directories are base locations that BitBake is looking for its recipes and configuration. For detailed explanations for each location, refer BitBake, Poky and Yocto documents. Nonetheless, I want to add a short notice that classes folder contains the definitions of base tasks such as *fetch, unpack, patch, build* and others, conf folder has the definitions of global variables and basic configuration variables that are used in BitBake, recipes-slos folder has BitBake recipes for setting up the toolchain and for build SLOS.

3) In this step, let's reuse the basic tasks defined in vanilla BitBake not in Poky. Poky

already has many tasks defined in its meta layers but we don’t need them and don’t want them. Go to BitBake repository cloned in step 1 and copy the *base.bbclass* file from *classes* directory to our *meta-slos/classes*. This base.bbclass file defines only very basic tasks. Open it and skim through the base.bbclass file. We have to add our own tasks into this file later to build SLOS.

4) Copy the *conf/bitbake.conf* from vanilla BitBake to *meta-slos/conf.* This file has definitions of BitBake environment variables. Open it and skim through those variables.

5) Create a file *conf/layer.conf* file and add following lines.

```
BBPATH .= ":${LAYERDIR}"
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb"
BBFILE_COLLECTIONS += "slos"
BBFILE_PATTERN_slos = "^${LAYERDIR}/"
BBFILE_PRIORITY_slos = "5"
```

These lines are needed for BitBake to find the recipe files, where they are, what is their file extension. I cited the meanings of each variable from BitBake user manual as it is.

BBPATH
Used by BitBake to locate class (.bbclass) and configuration (.conf) files. This variable is analogous to the PATH variable.

BBFILES
List of recipe files BitBake uses to build software.

BBFILE_COLLECTIONS
Lists the names of configured layers. These names are used to find the other BBFILE_* variables. Typically, each layer appends its name to this variable in its conf/layer.conf file.

BBFILE_PATTERN
Variable that expands to match files from BBFILES in a particular layer. This variable is used in the conf/layer.conf file and must be suffixed with the name of the specific layer (e.g. BBFILE_PATTERN_emenlow).

BBFILE_PRIORITY
Assigns the priority for recipe files in each layer.

This variable is useful in situations where the same recipe appears in more than one layer. Setting this variable allows you to prioritize a layer against other layers that contain the same recipe - effectively letting you control the precedence for the multiple layers. The precedence established through this variable stands regardless of a recipe's version (PV variable). For example, a layer that has a recipe with a higher PV value but for which the BBFILE_PRIORITY is set to have a lower precedence still has a lower precedence.

A larger value for the BBFILE_PRIORITY variable results in a higher precedence. For example, the value 6 has a higher precedence than the value 5. If not specified, the BBFILE_PRIORITY variable is set based on layer dependencies (see the LAYERDEPENDS variable for more information. The default priority, if unspecified for a layer with no dependencies, is the lowest defined priority + 1 (or 1 if no priorities are defined).

6) Let’s add our first dummy recipe. Make a directory named *toolchain* in the *meta-slos/recipes-slos* directory *(meta-slos/recipes-slos/toolchain)*. Create *toolchain_0.1.bb* and input following lines.

```
DESCRIPTION = "Install gcc toolchain to build SLOS"
PR = "r1"
PV = "1.0"
SRCREV = "v1.0"
```

These don’t do any meaningful actions but setting up the version variables. Variable *PR* is for recipe revision and *PV* is for recipe version. *SRCREV* is source code revision.

7) Now, let’s build our meta-slos layer for the first time. Run following command in Poky directory.

```
. oe-init-build-env build-slos
```

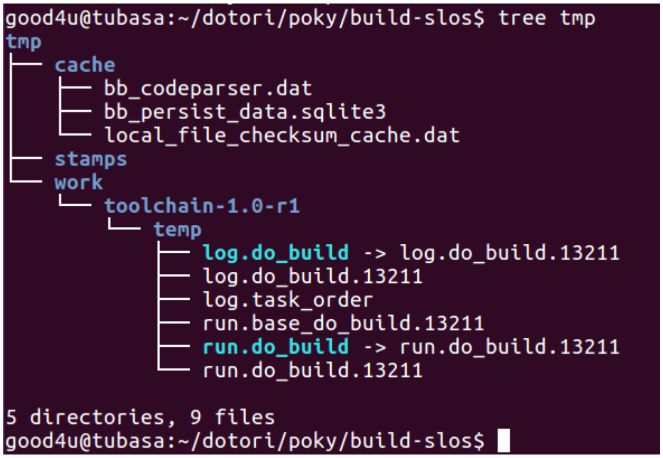
You can use *source* instead of dot (.) in the command. This command sets up build directory and make current directory to the build directory; in this example it is *build-slos.* Then, open the *conf/bblayers.conf* file in the build-slos directory and replace the BBLAYERS variable with your meta-slos path. Following is an example.

```
BBLAYERS ?= " \
        /home/good4u/dotori/poky/meta-slos “
```

We are ready to build meta-slos. Run *bitbake world -vDDD.* Then you will see output messages displayed in your screen. Currently, there is not meaningful tasks, BitBake performs a default task in the base.bbclass which is a *build* task. Following messages will be printed at the end of the command.

```
NOTE: Executing Tasks
DEBUG: Stampfile /home/good4u/dotori/poky/build-slos/tmp/stamps/toolchain-1.0-r1.do_build not available
DEBUG: Parsing /home/good4u/dotori/poky/meta-slos/recipes-slos/toolchain/toolchain_0.1.bb (full)
DEBUG: Executing task do_build
DEBUG: toolchain-1.0-r1 do_build: Executing python function do_build
DEBUG: toolchain-1.0-r1 do_build: Executing python function base_do_build
DEBUG: toolchain-1.0-r1 do_build: Python function base_do_build finished
DEBUG: toolchain-1.0-r1 do_build: Python function do_build finished
DEBUG: Teardown for bitbake-worker
NOTE: Tasks Summary: Attempted 1 tasks of which 0 didn't need to be rerun and all succeeded.
```

Notice the *tmp* directory in build-slos contains all outputs of BitBake build. Source git repositories, tools, build object files and all others are located below this directory. Currently, the tmp directory has only log files and shows as below.



```
good4u@tubasa:~/dotori/poky/build-slos$ tree tmp
tmp
├── cache
│   ├── bb_codeparser.dat
│   ├── bb_persist_data.sqlite3
│   └── local_file_checksum_cache.dat
├── stamps
└── work
    └── toolchain-1.0-r1
        └── temp
            ├── log.do_build -> log.do_build.13211
            ├── log.do_build.13211
            ├── log.task_order
            ├── run.base_do_build.13211
            ├── run.do_build -> run.do_build.13211
            └── run.do_build.13211

5 directories, 9 files
good4u@tubasa:~/dotori/poky/build-slos$
```

Figure A-5: Files in the tmp directory

8) We will add our tasks into the base.bbclass file. This file defines the basic tasks which are inherited to recipe files by default. If recipe doesn't want a task in the base.bbclass, it can overwrite the base task with its own task.
base.bbclass file defines *do_build* task and all it does is just print some notes. We need the BitBake to do some meaningful tasks such as setting up the cross compiler toolchain, fetching SLOS sources from the github repository, checking it out and finally compiling the SLOS. For this, we will copy those task definitions from the

meta/class/base.bbclass file. This file is already used, verified in Poky. First, let’s add a task to set up the toolchain. Copy *fetch, unpack* task from *meta/class/base.bbclass* and paste it to the end of our base.bbclass file. Add *addtask fetch before do_unpack* line before fetch task. This makes fetch task run before unpack task. Edit the last line as *EXPORT_FUNCTIONS do_fetch do_unpack do_build.* I removed unnecssary checksum python functions, but if you want them, just copy it from the Poky’s base.bbclass. The last part of our base.bbclass file looks like below.

```
addtask fetch before do_build
do_fetch[dirs] = "${DL_DIR}"
do_fetch[file-checksums] = ${@bb.fetch.get_checksum_file_list(d)}
do_fetch[file-checksums] += " ${@get_lic_checksum_file_list(d)}"
do_fetch[vardeps] += "SRCREV"
python base_do_fetch() {

src_uri = (d.getVar('SRC_URI') or "").split()
if len(src_uri) == 0:
         return
try:
         fetcher = bb.fetch2.Fetch(src_uri, d)
         fetcher.download()
 except bb.fetch2.BBFetchException as e:
         bb.fatal(str(e))
 }

 addtask unpack after do_fetch before do_build
 do_unpack[dirs] = "${WORKDIR}"
 do_unpack[cleandirs] = "${@d.getVar('S') if os.path.normpath(d.getVar('S')) != os.path.normpath(d.getVar('WORKDIR')) else os.path.join('${S}', 'patches')}"

 python base_do_unpack() {
 src_uri = (d.getVar('SRC_URI') or "").split()
 if len(src_uri) == 0:
         return

 try:
     fetcher = bb.fetch2.Fetch(src_uri, d)
     fetcher.unpack(d.getVar('P'))
 except bb.fetch2.BBFetchException as e:
        bb.fatal(str(e))
  }

  EXPORT_FUNCTIONS do_fetch do_unpack do_build
```

9) We will add a BitBake recipe for toolchain in this step. First, download the toolchain tar file *(gcc-arm-none-eabi-6-2017-q1-update-linux.tar.bz2)* from ARM developer site. If you can't find it, you can download it from my *tools* repository (*github.com/chungae9ri/tools)*. Make a *files* directory in a *toolchain* directory and copy the gcc toolchain into the *toolchain/files* directory.
Open previous toolchain_0.1.bb file and add following lines.

```
FILESEXTRAPATHS_prepend := "${THISDIR}/files:"
SRC_URI = "\
        file://gcc-arm-none-eabi-6-2017-q1-update-linux.tar.bz2"
TOOLCHAIN_OUT = "${HOME}/bin/arm-2017q1"

do_tc[nostamp] = "1"
do_tc() {
mkdir -p ${TOOLCHAIN_OUT}
cp ${S}/gcc-arm-none-eabi-6-2017-q1-update/arm-none-eabi ${TOOLCHAIN_OUT}/ -r
cp ${S}/gcc-arm-none-eabi-6-2017-q1-update/bin ${TOOLCHAIN_OUT}/ -r
cp ${S}/gcc-arm-none-eabi-6-2017-q1-update/lib ${TOOLCHAIN_OUT}/ -r
cp ${S}/gcc-arm-none-eabi-6-2017-q1-update/share ${TOOLCHAIN_OUT}/ -r
}

do_build() {
        :
 }

addtask tc after do_unpack before do_build
```

The *do_fetch(), do_unpack()* tasks we copied from Poky are doing uncompress it to build-slos directory. The tar file is defined as FILESEXTRAPATHS_prepend and SRC_URI variables. Fetch and unpack task uses this variable to find the source file. Then, we added our custom task *do_tc()* task which just makes a directory and copies the toolchain files to that. Finally, we add our *tc* task running before do_build but after do_unpack. *do_tc[nostamp] = "1"* makes the *do_tc()* task run whenever BitBake runs even though there is no changes in the files.

10) Run *bitbake world -vDDD* in the build-slos directory. BitBake reads recipes, interprets them and executes the tasks defined. The defined sequence of tasks should be do_fetch() -> do_unpack() -> do_tc() -> do_build(). Check *${HOME}/bin/* directory has the *arm-2017q1* directory and it contains gcc arm toolchain directories.

11) From this step, we will add the recipe to build SLOS. Let's add a recipe for SLOS. Create a *slos* directory in *meta-slos/recipes-slos,* and create a file named *slos_0.1.bb.* This file stores the recipes for BitBake to build SLOS. First, add following line into the recipe.

```
DESCRIPTION = "BitBake recipe slos build"
PR = "r1"
PV = "1.0"
SRCREV = "v1.0"
SRC_URI = "git://github.com/chungae9ri/slos;protocol=https"
```

The *SRC_URI* is used for *do_fetch()* to git clone from the repository and *do_unpack()* task checks out the master branch. All these are done in the *tmp* directory. Then, we are going to borrow some implementations from Poky's *base.bbclass*. Copy and add following lines into the recipe.

```
export MAKE = "make"
EXTRA_OEMAKE = ""
EXTRA_OECONF = ""

oe_runmake_call() {
    bbnote ${MAKE} ${EXTRA_OEMAKE} "$@"
    ${MAKE} ${EXTRA_OEMAKE} "$@"
}

oe_runmake() {
    oe_runmake_call "$@" || die "oe_runmake failed"
}

do_build[dirs] = "${B}/git"
do_build[nostamp] = "1"
do_build() {
    cd ${B}/git
```

```
     if [ -e Makefile -o -e makefile -o -e GNUmakefile ]; then
                oe_runmake || die "make failed"
     else
                bbnote "nothing to compile"
     fi
 }
 addtask build after do_tc

 addtask clean
 do_clean[dirs] = "${B}/git"
 do_clean[nostamp] = "1"
 do_clean() {
     cd ${B}/git

     if [ -e Makefile -o -e makefile -o -e GNUmakefile ]; then
                oe_runmake clean || die "make failed"
     else
       bbnote "nothing to clean"
fi
}
```

do_build() task first goes to the SLOS source directory and calls *oe_runmake()* function which is used in Poky. It is same as *Make* and you can use *make* instead of calling *oe_runmake.* Another task in this recipe is *do_clean()* task which calls the *clean* target in the SLOS Makefile.

12) Let’s run BitBake build for SLOS. go to the build-slos directory, and remove the *tmp* directory for test. Then, run *bitbake world -vDDD.* BitBake will first set up the toolchain by uncompressing the gcc tar ball and copies the binary folders to ${HOME}/bin/arm-2017q1 directory. Then, it clones the SLOS source and builds with the toolchain set in the previous step. All these steps are done in one command.

A4. Summary

We just experienced a simple BitBake build process by making the *meta-slos.* Yocto is much more complicated build steps as in figure A-2. All steps (fetching, patching, building, packaging etc.) in A-2 are done automatically. Nonetheless, by developing the meta-slos from scratch, we can get the basic ideas on BitBake and Yocto.