« 高通MSM8939出問題 聯發科MT6752迎來春天 64位A53架構發威：高通新驍龍410 MSM8916解析

ARMv8 與 Linux的新手筆記

ARMv8 與 Linux的新手筆記

by lod

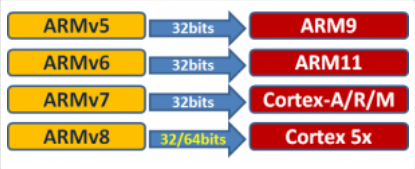
hlchou@gmail.cor

從iPhone 5S採用ARMv8處理器架構後,對於ARMv8 64bits的相關討論很多,也受到大家關注,Google也如預期在2014年底前推出了Android Lollipop (也就是Android 5.0) 操作環境.(官方網站 **http://www.android.com/versions/lollipop-5-0/**) ,這也是目前第一個同時支援32bits與64bits執行環境的Android操作環境,而面對新的ARMv8 64bits技術,目前主要還是以ARM本身所公布的技術資料為主,筆者也會基於ARM與Google Android正是公開的技術資料為主,來跟大家介紹ARMv8相關的軟體開發技術,並會佐以Linux Kernel Source Code做對照,讓大家除了知道技術名詞外,也可以實際的在代碼上有所掌握,,期待這能對產業軟體開發者有一點點幫助.

首先,讓我們簡單的回顧一下ARM處理器的歷史,簡單的區分, 像是ARMv5核心架構主要用於ARM9處理器系列,而ARMv6的核心架構,則用於ARM11系列產品,雖然前述處理器架構都是32bits的指令,但為了更節省記憶體的需求,ARM也推出了16bits Thumb指令,用以在記憶體受限的環境下,可以在些微影響效能的前提,達到兩者的平衡,隨後又延伸這個需求,提供 16/32bits 效率更高的Thumb2指令集架構. 再來就是,近期智慧型手機最主流的Cortex-A系列處理器,則採用的是32bits ARMv7的處理器架構技術,而在導入64bits ARMv8架構前,也開始有一些更大記憶體需求的產品應運而生,ARM提供給32bits架構不超過4GB記憶體限制的解決方案就是LPAE(Large Physical Address Extension)技術,用以支援最大40bit的實體記憶體定址範圍,然在這機制下,每個Process所見的記憶體空間仍受限於32bits 4GB定址.

本文所主要討論的ARMv8架構,同時支援了 32bits 與 64bits兩個模式,並在32bits模式下,也向後相容之前ARMv7架構32bits的軟體產品,藉此可提供目前現有ARM 32bits架構的產品一個平順過渡到ARMv8 32/64bits架構的無縫接軌. ARM並支援在ARMv8 64bits的產品上,同時執行32bits的既有軟體模塊與針對最新64bits軟體架構所設計的軟件,兩個不同指令集與記憶體框架的行程可以同時執行,並透過系統提供的IPC(Inter-Process Communication) 進行協同工作.

參考ARM網頁(**http://www.arm.com/zh/products/processors/cortex-a/cortex-a53-processor.php**),目前已知對外公開的ARMv8架構包含了Cortex A53與A57,這兩者可用於配搭 big.LITTLE的架構,或可單獨採用.



如下表,為大家常見的ARMv7與ARMv8處理器的簡要比較,供參考.

CPU	Core Version	Pipeline	DMIPS/MHz	Physical Memory Addresses
Cortex A7	ARMv7	In-Order	1.9	LPAE 40bits
Cortex A15	ARMv7	Out-of-Order	3.5	LPAE 40bits
Cortex A53	ARMv8	In-Order	2.3	40bits
Cortex A57	ARMv8	Out-of-Order	4.1	44bits

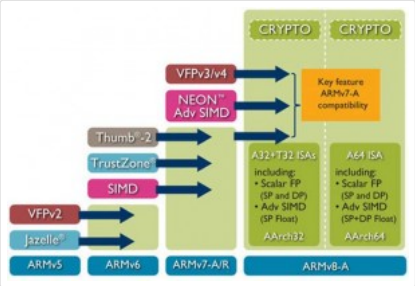
由於本文會有不少ARMv8 與 32/64bits相關技術名詞引用,為了便於訊息的一致性,筆者在此先定義如下

Items	Explain
AArch64	指基於64bits運作的ARMv8 Architecture.(General Purpose Registers X0-X30)
AArch32	指基於32bits運作的ARMv8 Architecture.並且相容於原本的ARMv7 Architecture. (General Purpose RegistersR0-R15)
A64	指在AArch64模式下支援的 ARM 64bits 指令集.
A32	指ARMv7架構下支援的 ARM 32bits 指令集,在ARMv8中也有新加入的A32指令集.
T32	指ARMv7架構下支援的 Thumb2 16/32bits指定集,在ARMv8中也有新加入的T32指令集.

本文所引用的軟體代碼,會以筆者撰寫本文時參考的Linux Kernel 3.16.3 為基礎,線上瀏覽的Source Code網址可以參考 **http://hala01.com/src/linux/linux-3.16.3/HTML/** .最後,撰寫時雖盡力確保資料無誤,若仍有所疏失還請多加包含.

ARMv8的基礎認識

目前的理解,談到ARMv8最多人引用的圖會是ARM網站(http://www.arm.com/zh/products/processors/instruction-set-architectures/index.php)所提供如下的架構示意圖(**http://www.arm.com/zh/images/roadmap/V5_to_V8_Architecture.jpg**). 簡要來說, ARMv8的架構沿襲以往ARMv7 與之前處理器技術的基礎，除了有既有16/32bits Thumb2指令的支援外，也向前相容現有的ARM 32bits指令集. 基於64bits的AArch64架構,除了新增A64 (ARM 64bits)指令外,也擴充現有的A32 (ARM 32bits) 與T32 (Thumb2 32bits)指令編碼可在ARMv8 AArch32架構中執行,因此若有特別針對ARMv8 32bits指令集新增指令所撰寫的程式碼(例如: 32bits Crypto),這類程式將只能在ARMv8或之後新的處理器架構上執行,而無法向前相容於ARMv8之前的32bits實體處理器執行上.



AArch64 表示為支援64bits Execution State,而AArch32則是支援 ARMv8之前的32bits Execution State,並有因應AArch64新增額外的能力,確保相容於ARMv7-A的架構. AArch32/64也都支援 SIMD (Single-Instruction Multiple-Data)與Floating-Point指令,其中的差異是,AArch32的SIND/Floating-Point使用64bits的暫存器,而AArch64使用的是128bits暫存器.

參考文件 Procedure Call Standard for the ARMR Architecture(**http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf**) 與 Procedure Call Standard for the ARM 64-bit Architecture(AArch64) (**http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf**),其中有關General purpose registers and AAPCS64 usage段落有針對這31個General Purpose 64bits暫存器的使用方式,可知64bits提供的額外暫存器,包括在Function Parameter/Return值處理,以及可供函式內優化使用的暫存器數量都增加,善用這些額外暫存器可減少對外部記憶體存取的頻率,並讓編譯器優化時,更有空間去改善執行效能.

AArch64 Register	Special	Role in the procedure call standard
x0…x7		Parameter/result registers
x8		Indirect result location register
x9…x15		Temporary registers
x16	IP0	The first intra-procedure-call scratch register (can be used by call veneers and PLT code); at other times may be used as a temporary register.
x17	IP1	The second intra-procedure-call temporary register (can be used by call veneers and PLT code); at other times may be used as a temporary register.
x18		The Platform Register, if needed; otherwise a temporary register.
x19…x28		Callee-saved registers
x29	FP	The Frame Pointer
x30	LR	The Link Register
SP		The Stack Pointer.

AArch32 Register	Special	Role in the procedure call standard
r0…r3		Parameter/result registers
r4…r8 **r9 (also as platform register)** **r10,r11**		Temporary registers
r12	IP	The Intra-Procedure-call scratch register.
r13	SP	The second intra-procedure-call temporary register (can be used by call veneers and PLT code); at other times may be as a temporary register.
r14	LR	The Platform Register, if needed; otherwise a temporary register.
r15	PC	Callee-saved registers

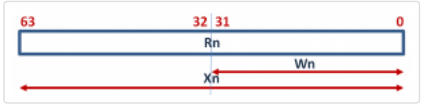
如下表,筆者嘗試從支援的暫存器/指令來分類AArch32 與 AArch64,以供簡要參考.

Execution State	Note
AArch64	1,提供**31個64bits**寬的General-Purpose Registers (from x0~x30,其中 x30亦可作為Procedure Link Registers) 2,提供64bits Program Counter(PC), Stack-Poiner(SP)與Exception-Link-Register (ELR) 3,提供**32個128bits**寬的SIMD Vector 與 Scalar Floating-Point暫存器 4,定義ARMv8 EL0~EL3共4個Execution Privilege **5, 支援64bits Virtual-Addressing** 6, 定義一組PSTATE用以保存PE(Processing Element)狀態.
AArch32	1,提供**16個32bits**寬的General-Purpose Registers (from r0~r12, 其中r13=SP, r14=LR and r15=PC, 並且r14需同時供ELR與Procedure Link Registers之用) 2, 提供一個ELR,用以作為從Hyp-Mode的Exception返回之用. 3, 提供**32個64bits**寬的Advanced SIMD Vector 與 Scalar Floating-Point暫存器 4, 提供A32與T32兩種指令集的組態 **5, 使用32bits Virtual-Addressing** 6, 只使用CPSR (Current Program State Register)保存PE(Processing Element)狀態.

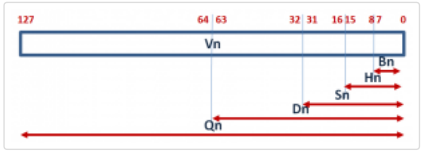
ARMv8共支援以下幾種Data types,

Data Type	Length
Byte **(B)**	8 bits.
Halfword **(H)**	16 bits.
Word **(S)**	32 bits.
Doubleword **(D)**	64 bits.
Quadword **(V)**	128 bits.

而ARMv8通用暫存器.可區分32bits (Wn)與 64bits (Xn)兩類,可供程式執行依需求使用.



至於SIMD/浮點數的部分,參考下圖可區分為8bits~128bits(16bits Half-Precision, 32bits Single-Precision and 64bits Double-Precision)不同的Size,並相容於IEEE 754.



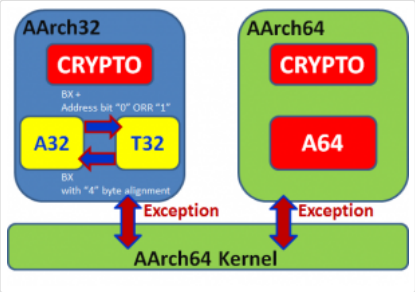
ARMv8跟之前ARM處理器相比,最大的亮點之一就是Crypto加密指令集的支援,目前這部份的支援主要包括基於SIMD指令的 AES,SHA1與SHA2-256硬體加速指令,可參考如下簡表.

Crypto	Explain

New ARMv32 Instruction	
AESD.8	AES single round decryption.
AESE.8	AES single round encryption.
AESIMC.8	AES inverse mix columns.
AESMC.8	AES mix columns.
SHA1C.32	SHA1 hash update accelerator (choose).
SHA1M.32	SHA1 hash update accelerator (majority).
SHA1P.32	SHA1 hash update accelerator (parity).
SHA1H.32	SHA1 hash update accelerator (rotate left by 30).
SHA1SU0.32	SHA1 schedule update accelerator, first part
SHA1SU1.32	SHA1 schedule update accelerator, second part
SHA256H.32	SHA256 hash update accelerator.
SHA256H2.32	SHA256 hash update accelerator upper part.
SHA256SU0.32	SHA256 schedule update accelerator, first part
SHA256SU1.32	SHA256 schedule update accelerator, second part
VMULL.P64	Polynomial multiply long, AES-GCM acceleration 64×64 to 128-bit.

綜觀上述的介紹後,筆者嘗試把AArch32與AArch64兩個執行環境用如下圖加以說明,希望會比較好理解兩者的差異. 紅色部分為這次ARMv8新增的模塊,而黃色部分則為ARMv7既有支援的指令集範圍,可以看到在AArch32的架構下,A32與T32指令集間可以透過BX搭配Address bit 0 為1的方式由ARM Mode切換到Thumb Mode,並可透過BX指令從Thumb Mode切換回ARM Mode,兩者轉換的成本很低. 且ARMv8所支援的Crypto指令也同時在AArch32與AArch64模式下都有支援,可用於加速這兩類指令集模式下加解密指令效率.

而跟過去習知ARM Mode與Thumb Mode指令集切換最大的差異在於,在AArch32與AArch64兩者32bits/64bits執行模式切換時,目前ARMv8架構下只能透過觸發Exception的方式進行切換.也因此這表示一個應用程式撰寫時就必須決定自己是要處於32bits 或64bits的場景.如果希望可以享受到兩個模式下的好處,就必須要同時具備32bits Process與64bits Process,兩者之間再透過IPC(Inter-Process Communication)進行溝通.



ARMv8把Execution Privilege區分為EL0到EL3共4個Level,根據目前的架構,會由下層系統的Execution State決定上層系統所在的模式.若下層系統為32bits, 則上層就只能為32bits

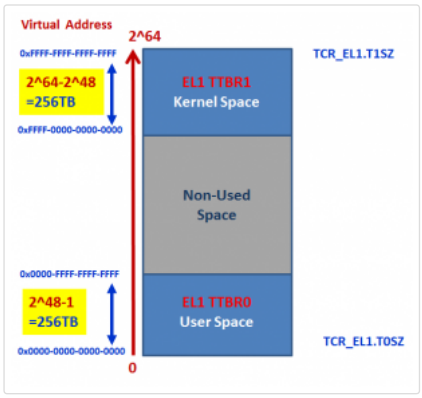
反之,若下層系統為64bits,上層就可選擇為32bits或64bits兩者之一,也因此若想要同時讓你的處理器軟體執行環境支援32bits Process與64bits Process,就必須要使用 64bits的Kernel執行環境.

把ARMv8處理器的基礎差異說明後,接下來讓我們進一步嘗試說明ARMv8 AArch32與AArch64在執行時期的Memory體系為何,藉此可對應用程式運作有更進一步的了解.

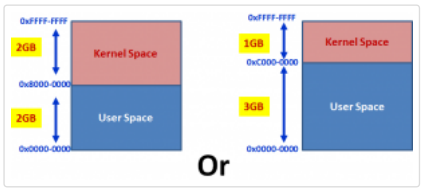
ARMv8 Memory Model

參考Linux Kernel 3.16.3(in /arch/arm/mm/proc-v7-2level.S, **http://hala01.com/src/linux/linux-3.16.3/HTML/S/19381.html**), ARM 32bits下會用TTBR0儲存當下User-Space行程所在的Page Table (也就是0xC0000000以下的記憶體空間),並用TTBR1儲存Kernel Space所在的Page Table (也就是0xC0000000以上的記憶體空間).

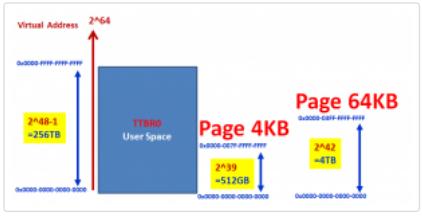
在ARMv8 64bits架構下,會透過EL1的TTBR0 (ttbr0_el1, in /arch/arm64/mm/proc.S, **http://hala01.com/src/linux/linux-3.16.3/HTML/S/22828.html**)儲存當下User-Space行程所在的Page Table,與EL1的TTBR1儲存Kernel Space所在的Page Table,並會依據Page Size與32/64bits行程而有不同的記憶體空間配置. 參考如下圖所示



如下為原本大家習知32bits Linux Kernel記憶體定址方式,依據需求開發者可以配置為 User與Kernel Space各2GB的Layout,或是修改為 User與Kernel Space分別為3GB與1GB的定址方式.



但由於ARM 64bits Kernel的分頁(Page)有分4KB 與 64KB兩種大小,參考 TASK_SIZE_64 (/arch/arm64/include/asm/memory.h,http://hala01.com/src/linux/linux-3.16.3/HTML/S/22730.html#L62)所指定的數值,其中當分頁大小為4KB時,決定TASK_SIZE_64大小的VA_BITS會等於 39,也就是2^39大小的Task空間(=512GB),若分頁大小為64KB時,則TASK_SIZE_64對應的VA_BITS會等於 42,也就是2^42大小的Task空間(=4TB).

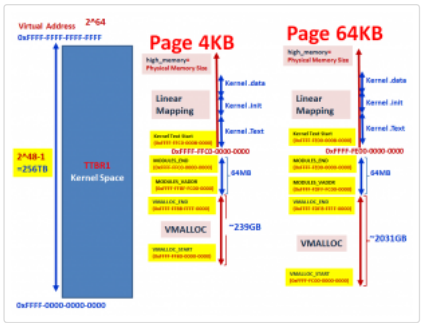


同時, Kernel Space的範圍會透過 PAGE_OFFSET=(UL(0xffffffffffffffff) << (VA_BITS – 1))取得(in /arch/arm64/include/asm/memory.h , http://hala01.com/src/linux/linux-3.16.3/HTML/S/22730.html#L49), 且ARM64 Linux Kernel下的high_memory會等於裝置所配置的實體記憶體大小加上PAGE_OFFSET,這塊記憶體可用於線性Memory Mapping實體記憶體與核心虛擬記憶體空間. 若有配置Sparse Memory Virtual memmap用以優化 pfn_to_page/page_to_pfn 流程,則vmemmap 起點為Linear Mapping記憶體的起點 virt_to_page(PAGE_OFFSET)終點會等於virt_to_page(high_memory).

當分頁大小為4KB時,決定PAGE_OFFSET大小的VA_BITS會等於 39,因此 PAGE_OFFSET=(UL(0xffffffffffffffff) << (39 – 1))= 0xFFFFFFC000000000,而Kernel Driver所在的區間 MODULES_END =PAGE_OFFSET=0xFFFFFFC000000000 與 MODULES_VADDR=MODULES_END – SZ_64M=0xFFFFFFBFFC000000, 並且VMALLOC_START= (UL(0xffffffffffffffff) << VA_BITS)= (UL(0xffffffffffffffff) << 39)= 0xFFFFFF8000000000,與VMALLOC_END=(PAGE_OFFSET – UL(0x400000000) – SZ_64K)= 0xFFFFFFBBFFFF0000.

若分頁大小為64KB時,則PAGE_OFFSET對應的VA_BITS會等於 42,也就是PAGE_OFFSET=(UL(0xffffffffffffffff) << (42 – 1))= 0xFFFFFE0000000000, 而Kernel Driver所在的區間 MODULES_END =PAGE_OFFSET=0xFFFFFE0000000000與 MODULES_VADDR=MODULES_END – SZ_64M=0xFFFFFDFFFC000000. 並且VMALLOC_START= (UL(0xffffffffffffffff) << VA_BITS)= (UL(0xffffffffffffffff) << 42)= 0xFFFFFC0000000000,與VMALLOC_END=(PAGE_OFFSET – UL(0x400000000) – SZ_64K)= 0xFFFFFDFBFFFF0000.

有關Kernel Memory Mapping的概念,可參考如下示意圖



秉持一貫有Source Code有真相的原則,讓我們從Linux Kernel Source Code的角度來進一步的說明原委,首先從ARM64架構下一個應用程式從執行檔開始貼到記憶體布局的流程來加以說明,

1, 當使用者或應用載入一個新的程式時(例如呼叫load_elf_binary),就會透過函式setup_new_exec(in /fs/exec.c, **http://hala01.com/src/linux/linux-3.16.3/HTML/S/8148.html#L1101**)來為新的Process的記憶體布局進行配置.

2, 在函式setup_new_exec中,則會呼叫arch_pick_mmap_layout(current->mm);進入函式arch_pick_mmap_layout(in /arch/arm64/mm/mmap.c, **http://hala01.com/src/linux/linux-3.16.3/HTML/S/22812.html#L84**),依據不同的處理器差異**(ARM 32bits/64bits,MIPS,Parisc,PowerPC,S390,Sparc,Tile,x86,..etc)**,為目前的Process 記憶體布局進行配置

3, 由於本文著重於ARM 64bits,因此會以ARM64中的函式 arch_pick_mmap_layout(in /arch/arm64/mm/mmap.c, **http://hala01.com/src/linux/linux-3.16.3/HTML/S/22812.html#L84**)為主加以說明,首先這函式可以分為兩個部分

```
void arch_pick_mmap_layout(struct mm_struct *mm)
{
```

if (mmap_is_legacy()) {

mm->mmap_base = TASK_UNMAPPED_BASE;

mm->get_unmapped_area = arch_get_unmapped_area;

} **else** {

mm->mmap_base = mmap_base();

mm->get_unmapped_area = arch_get_unmapped_area_topdown;

}

}

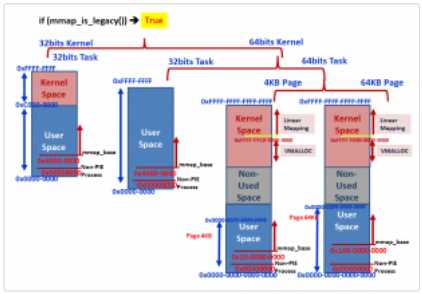
3.1, **若if (mmap_is_legacy())成立**è

此時Task的mm->get_unmapped_area為arch_get_unmapped_area(in /mm/mmap.c, **http://hala01.com/src/linux/linux-3.16.3/HTML/S/9939.html#L1873**),表示該Task Process每一個新的Memory Mapping (包括應用程式本身,動態函式庫,MMAP配置…等),都會由低位址往高位址依序配置而上,這比較像是最早對於原本對於MMAP的概念,也就是Legacy的作法.

對應到 32bits Kernel, Task 的mm->mmap_base會等於TASK_UNMAPPED_BASE,若此時Kernel Space起點為0xC0000000,則會從0x40000000為起點.

若是 64bits Kernel, 則 32bits Task的TASK_UNMAPPED_BASE 會等於(PAGE_ALIGN(TASK_SIZE / 4))(in /arch/arm64/include/asm/memory.h, **http://hala01.com/src/linux/linux-3.16.3/HTML/S/22730.html#L65**),其中因為TASK_SIZE_32為0x100000000,因此會跟32bits Kernel下的32bits Task一樣都以0x40000000為起點.

若為64bits Kernel下的64bits Task, 由於 TASK_SIZE_64 等於 (UL(1) << VA_BITS),若該64bits Kernel使用4KB Page為單位,則VA_BITS 等於39 (若使用64KB Page為單位,則VA_BITS 等於42),對應到TASK_SIZE_64 等於(UL(1) << VA_BITS),因此在4KB Page設定下, TASK_UNMAPPED_BASE 透過(PAGE_ALIGN(TASK_SIZE / 4))會等於0x2000000000 (若使用64KB Page, 則為0x10000000000), 由於這些組合比較多,筆者把上述的組合繪製如下圖所示



3.2,**若if (mmap_is_legacy()) 不成立**è

此時Task的mm->get_unmapped_area為arch_get_unmapped_area_topdown(in /mm/mmap.c, **http://hala01.com/src/linux/linux-3.16.3/HTML/S/9939.html#L1909**),表示該Task Proces每一個新的Memory Mapping (包括應用程式本身,動態函式庫,MMAP配置…等),都會反過來,由高位址往低位址依序配置而下,這是屬於目前比較新的Android手機環境所要求的作法. 在這架構上, Task 的mm->mmap_base會透過呼叫函式 **“unsigned long mmap_base(void)”** (in /arch/arm64/mm/mmap.c , **http://hala01.com/src/linux/linux-3.16.3/HTML/S/22812.html#L68**) 決定.

如下為函式mmap_base的實作

static unsigned long mmap_base(void)

{

unsigned long gap = rlimit(RLIMIT_STACK);

if (gap < MIN_GAP)

gap = MIN_GAP;

else if (gap > MAX_GAP)

gap = MAX_GAP;

return PAGE_ALIGN(STACK_TOP – gap – mmap_rnd());

}

其中, gap = rlimit(RLIMIT_STACK) = rlimit(3)會透過ACCESS_ONCE(tsk->signal->rlim[limit].rlim_cur) 取得目前系統的配置數值.

參考如下代碼, 其中, MAX_GAP會等於 (STACK_TOP/6*5),以 ARM 64bits下的4KB Page來說這個值為(TASK_SIZE_64/6*5) = ((UL(1) << VA_BITS)/6*5)=0x6AAAAAAAA9 (in arch/arm64/mm/mmap.c,http://hala01.com/src/linux/linux-3.16.3/HTML/S/22812.html#L37),而 MIN_GAP會等於 (SZ_128M + ((STACK_RND_MASK << PAGE_SHIFT) + 1)) 以 ARM 64bits下的4KB Page來說這個值為(0x08000000 + (((0x3ffff >> (12 – 12)) << 12) + 1))=0x47FFF001 (in arch/arm64/mm/mmap.c,http://hala01.com/src/linux/linux-3.16.3/HTML/S/22812.html#L36)

/*

*** Leave enough space between the mmap area and the stack to honour ulimit in**

*** the face of randomisation.**

***/**

#define MIN_GAP (SZ_128M + ((STACK_RND_MASK << PAGE_SHIFT) + 1))

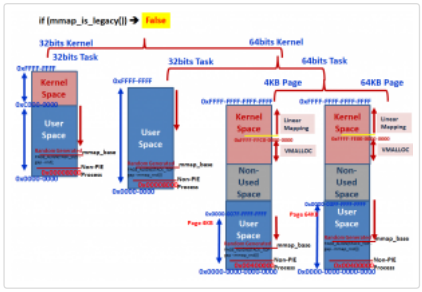
#define MAX_GAP (STACK_TOP/6*5)

在ARM 64bits搭配4KB Page配置下, 函式mmap_rnd 實作(in arch/arm64/mm/mmap.c ,http://hala01.com/src/linux/linux-3.16.3/HTML/S/22812.html#L58),其中 rnd = (long)get_random_int() & (STACK_RND_MASK >> 1) , 而get_random_int函式會透過MD5 Hash機制回傳一個整數任意值, STACK_RND_MASK會等於 (0x3ffff >> (PAGE_SHIFT – 12)) =0x3ffff (in /arch/arm64/include/asm/elf.h ,http://hala01.com/src/linux/linux-3.16.3/HTML/S/22803.html#L157).

因此, rnd 會等於整數隨機任意值 & 0x3ffff,之後再進行rnd << (PAGE_SHIFT + 1)等於 rnd << (12 + 1)= 0x3ffff <<13 =0x7fffe000.

```
/*
* Since get_random_int() returns the same value within a 1 jiffy window, we
* will almost always get the same randomisation for the stack and mmap
* region. This will mean the relative distance between stack and mmap will be
* the same.
* To avoid this we can shift the randomness by 1 bit.
*/
static unsigned long mmap_rnd(void)
{
unsigned long rnd = 0;
if (current->flags & PF_RANDOMIZE)
rnd = (long)get_random_int() & (STACK_RND_MASK >> 1);
return rnd << (PAGE_SHIFT + 1);
}
```

函式mmap_base 最後回傳的PAGE_ALIGN(STACK_TOP – gap – mmap_rnd()) = AGE_ALIGN(0x7fffffffff – gap – 0x7fffe000),對應到實際的產品上,會是如下的Layout



接下來,讓我們從CPRS與Process State來進一步了解ARMv8的系統程式.

從 CPSR 看 PSTATE (Process state)

要了解ARMv7 與ARMv8 的System Model最好方式就是從 Current Program Status Register (CPSR) 來看兩者的差異,首先筆者參考 ARMv8 ARM文件 (**http://www.cs.utexas.edu/~peterson/arm/DDI0487A_a_armv8_arm_errata.pdf**) 與ARMv7 Architecture Reference Manual的文件(**http://liris.cnrs.fr/~mmrissa/lib/exe/fetch.php?media=armv7-a-r-manual.pdf**), 並以此為基礎來加以說明.

根據ARM的文件, ARMv7 , ARMv8 32bits 與 ARMv8 64bits的CPSR相比有如下的差異. 最基礎來看就是ARMv7與ARMv8 32bits的比較,會看到ARMv8 32bits多了Bit 20:IL新的PSR屬性. 而ARMv8 32bits與ARMv8 64bits相比則是少了IT與GE屬性,但卻多了D屬性(取代原本32bits下的E屬性).

ARMv8 AArch64	**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**
	N	**Z**	**C**	**V**	**RES0**						**SS**	**IL**	**RES0**										**D**	**A**	**I**	**F**	**RES0**	**0**	**M[3:0]**			
	Condition Flag																							**Mask bits**				**M[4]**				
ARMv8 AArch32	**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**
	N	**Z**	**C**	**V**	**Q**	**IT[1:0]**		**J**	**RES0**			**IL**	**GE[3:0]**				**IT[7:2]**						**E**	**A**	**I**	**F**	**T**	**1**	**M[3:0]**			
	Condition Flag																							**Mask bits**				**M[4]**				
ARMv7	**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**
	N	**Z**	**C**	**V**	**Q**	**IT[1:0]**		**J**	**Reserved RAZ/SBZP**				**GE[3:0]**				**IT[7:2]**						**E**	**A**	**I**	**F**	**T**	**M[4:0]**				
	Condition Flag																							**Mask bits**								

首先, 簡述PSR(Program Status Register) 程式狀態暫存器的意義,這個暫存器主要用來紀錄程序狀態之用,包括反映出目前所處的處理器模式,指令集狀態,以及反應出條件(Cond.)執行指令判斷執行的依據.舉個例子來說,在ARMv7 或ARMv8 32bits下, 當我們從CPSR的4-0bits取出值為b10111就可以知道目前所在的Exception Handler,是發生了Abort,之後再判斷SPSR的4-0bits,若為b10011(SVC Mode)或b10000(User Mode),就可以知道在觸發這個Abort前,處理器是在執行哪一個模式下的程式碼,如果擔心有因為Exception Handle設計不當導致的Abort重入問題,也可以透過CPSR/SPSR前後模式比對,知道是不是Abort重入,可以鎖定潛在的系統問題加以解決. 如下簡述每個欄位的意義

ARMv7			**ARMv8 AArch32**			**ARMv8 AArch64**	
功能	**說明**	**位元**	**功能**	**說明**	**位元**	**功能**	**說明**
	Mode Bits模式位元						Mode Bits模 0b0000 – E 0b0100 – E

Mode[3:0]	b10000(0×0010) -User Mode b10001(0×0011)- FIQ Mode b10010(0×0012)-IRQ Mode b10011(0×0013)-Supervisor Mode b10111(0×0017)-Abort Mode b11011(0x001b)-Undefined Mode b11111(0x001F)-System Mode b10110(0×0016)-Secure Monitor	**3-0**	Mode[3:0]	**the same as ARMv7**	**3:0**	Mode[3:0]	0b0101 – E 0b1000 – E 0b1001 – E 0b1100 – E 0b1101 – E **M[3:2] hol** **M[1] is un** **M[0] is us** **SP0 and 1** **EL.**
Mode[4]	1	**4**	Mode[4]	**= 1 = Exception taken from AArch32**	**4**	Mode[4]	**= 0 = Exce**
T	Thumb state bit 0=ARM 1=Thumb	**5**	T	**the same as ARMv7**	**5**	RES0	Reserved
F	FIQ Disable 1=禁止 0=允許	**6**	F	**the same as ARMv7**	**6**	F	FIQ Disabl 1=禁止 0=允許
I	IRQ Disable 1=禁止 0=允許	**7**	I	**the same as ARMv7**	**7**	I	IRQ Disabl 1=禁止 0=允許
A	Asynchronous data abort mask bit. 1=禁止 0=允許	**8**	A	**the same as ARMv7**	**8**	A	SError (Sys are: 1=禁止 0=允許
E	Endianness execution state bit. Controls the load and store endianness for data accesses: 0=Little-endian operation 1=Big-endian operation.	**9**	E	**the same as ARMv7** **但由於ARMv8有EL0~EL3,這個Bit可以用來表示目前所在的EL是支援Little or Big-endian**	**9**	D	Process st 0 = 允許 D Software s level . 1 = 禁止 D Software s level. (若除錯Tar 個Bit,所以當 Target EL.)
IT[7:2]	If-Then execution state bits for the Thumb IT (If-Then) instruction.	**10**	c	IT state bits	**19:10**	RES0	Reserved
		11	b				
		12	a				
		15-13	IT_cond				
GR[3:0]	Greater than or Equal flags	**19-16**	GR[3:0]	Greater than or Equal flags			
RAZ/SBZP	Reserved	**20**	IL	Illegal Execution State bit. Shows the value of PSTATE.IL immediately before the exception was taken.	**20**	IL	Illegal Exec immediatel taken.
		23:21	RES0	Reserved	**21**	SS	Software s when an ex
					27:22	RES0	Reserved
J	Jazelle State Bit	**24**	J	Jazelle State Bit			
IT[1:0]	If-Then execution state bits for the Thumb IT (If-Then) instruction.	**25**	e	IT state bits			
		26	d				
Q	Cumulative saturation bit.	**27**	Q	Sticky Overflow			
V	Overflow condition flag.	**28**	V	Overflow	**28**	V	Set to the v to EL1, and flag on exe
C	Carry condition flag	**29**	C	Carry/Borrow/Extend	**29**	C	Set to the v to EL1, and flag on exe
							Set to the v

Z	Zero condition flag	**30**	Z	Zero	**30**	Z	to EL1, and flag on exe
N	Negative condition flag	**31**	N	Negative/Less than	**31**	N	Set to the v to EL1, and flag on exe

若M[4]為1,則表示是在ARMv7原本的32bits Mode與ARMv8 AArch32下,我們可以直接透過M[4:0]判定目前是在哪個CPU State(User, SVC, Data Abort, Undef….etc)

但若M[4]為0則表示當下是在ARMv8 AArch64下,會變成改用M[3:0]中的M[3:2]可用以表示當下所在的Exception Level.

而M[0]若為0表示預設使用SP_EL0作為Stack Pointer,若1則預設使用每個Exception Level對應的SP_ELx作為Stack Pointer. 當所在的Exception Level為EL0,則此時SP(Stack Pointer)就會預設定義為SP_EL0,通常每個Exception Level都有其所對應的SP暫存器,例如SP_ELx暫存器就可用於儲存每個不同Exception Level所對定的專屬 SP 置存器,例如 EL1使用 SP_EL1, EL2使用SP_EL2而EL3使用SP_EL3. 在軟體執行的過程中,可以透過 “MSR SPSel, #Imm1”指令選擇要使用每個Exception Level對應的SP_ELx暫存器還是統一只使用SP_EL0暫存器作為當下的Stack Pointer.

針對Debug的需求,AArch64下可以透過 Syndrome Register在切換不同CPU Exception Level時,判定當下系統觸發Exception Level轉換的原因,同樣的,要有Code才有真相,因此讓我們參考 arch/arm64/kernel/entry.S (**http://hala01.com/src/linux/linux-3.16.3/HTML/S/22867.html**),可知在Linux Kernel EL1 Entry的實作,會透過讀取ESR_EL1當做Exception Mode的判定,

```
el1_sync:
kernel_entry 1
mrs     x1, esr_el1                     // read the syndrome register
lsr     x24, x1, #ESR_EL1_EC_SHIFT      // exception class
cmp     x24, #ESR_EL1_EC_DABT_EL1       // data abort in EL1
b.eq    el1_da
cmp     x24, #ESR_EL1_EC_SYS64          // configurable trap
b.eq    el1_undef
cmp     x24, #ESR_EL1_EC_SP_ALIGN       // stack alignment exception
b.eq    el1_sp_pc
cmp     x24, #ESR_EL1_EC_PC_ALIGN       // pc alignment exception
b.eq    el1_sp_pc
cmp     x24, #ESR_EL1_EC_UNKNOWN        // unknown exception in EL1
b.eq    el1_undef
cmp     x24, #ESR_EL1_EC_BREAKPT_EL1    // debug exception in EL1
b.ge    el1_dbg
b       el1_inv
```

如下所示為ARMv8 64bits下, ESR (Exception Syndrome Register)的格式內容,

Format of the ESR_ELx registers	**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**
	EC(Exception Class)						**IL**	**ISS (Instruction Specific Syndrome)**																							

筆者依據ARM文件簡述內容意義如下所示

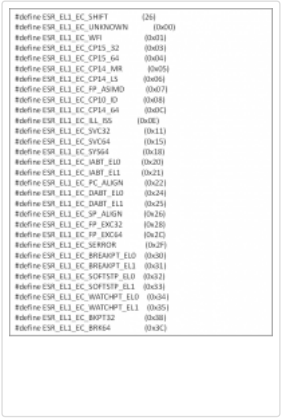
EC[31:26]: 表示Exception Class, 用以表示觸發這次Exception的原因 (例如從EL0->EL1,…etc)

IL[25]: 表示Instruction Length, 0表示 16bits Trapped-Instruction, 1表示32bits Trapped-Instruction.

è以Software Breakpoint Instruction而言, 0表示16bits T32(Thumb) BKPT指令觸發,1表示32bits A32(ARM) BKPT或A64 BRK指令觸發.

ISS [24:0]: 用以作為Instruction Specific Syndrome 欄位,可在AArch32或AArch64不同模式下,有不同欄位的顯示意義.

ESR_ELx是一個AArch64下CPU Mode基礎狀態判定的基礎,依據Exception Level的改動, AArch64有支援ESR_EL1, ESR_EL2, and ESR_EL3用以提供給 EL0->EL1, EL1->EL2與EL2->EL3模式轉換過程的CPU State判斷. 但簡單來說,最直覺的使用就是參與EC與IL欄位,依據目前3.16版本的Linux Kernel Source Code的arch/arm64/include/asm/esr.h (**http://hala01.com/src/linux/linux-3.16.3/HTML/S/22702.html**)來說,會根據EC定義如下的處理器模式組態作為EL0<->EL1的判斷之用.



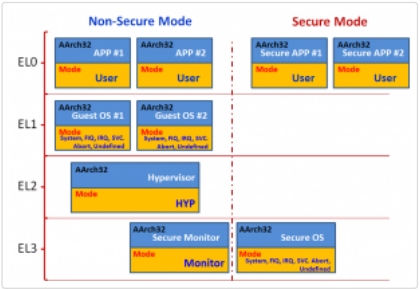
以Linux Kernel 的EL1 mode handlers來說共支援以下的Exception Class

Exception Class in Linux Kernel EL1	Notes
ESR_EL1_EC_UNKNOWN (0x00) **(unknown exception in EL1)**	Unknown reason (用以包括其它尚未被Exception Class包括到的Exception類型) 例如像是 1, 錯誤的指令集編碼 (像是 Undefined Instruction..etc),不管是在Debug or Non-Debug State 2, Attempted execution of: — An HVC instruction when disabled by HCR_EL2.HCD or SCR_EL3.HCE. — An SMC instruction when disabled by SCR_EL3.SMD. — An HLT instruction when disabled by EDSCR.HDE 3, Attempted execution of an MSR or MRS to SP_EL0 when the value of SPSel.SP is 0. 4, …(還有許多類型 就不一一敘述.)
ESR_EL1_EC_SYS64 (0x18) **(configurable trap)**	MSR, MRS, or System instruction execution, that is not reported using EC 0x00, 0x01, or 0 Exception from MSR, MRS, or System instruction execution in AArch64 state
ESR_EL1_EC_PC_ALIGN (0x22) **(pc alignment exception)**	Misaligned PC exception
ESR_EL1_EC_DABT_EL1 (0x25) **(data abort in EL1)**	Data Abort taken without a change in Exception level
ESR_EL1_EC_SP_ALIGN (0x26) **(stack alignment exception)**	Stack Pointer Alignment exception
ESR_EL1_EC_BREAKPT_EL1(0x31) **(debug exception in EL1)**	Breakpoint exception taken without a change in Exception level

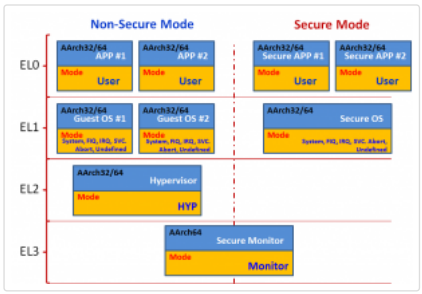
在簡述ARM Process State之後,讓我們進一步說明ARMv8 Security Model.

ARMv8 Security Model

針對Security的需求, ARMv8的系統軟體設計可以提供如下Secure-Mode與Non-Secure Mode的組態,如前面所提到的,若底層EL(Exception Level)為32bits,則上層EL的軟體就只能是32bits.



若底層的EL為64bits,則上層EL就可以依據需求選擇為 32bits或是64bits 的軟體模塊.



結語

在不久的未來,即將有許多配備ARMv8的Android/Linux裝置被普及到市面上,除了更好的執行效率外,64bits更大的記憶體定址能力,也讓原本32bits High/Low-Zone體系被改變,開發者也省去一些力氣在記憶體Zone區的優化上,但對使用者來說,隨著更多搭配64bits優化方案的日趨成熟,像是編譯器, ART執行環境或第三方應用軟體的優化,相信所帶來的使用者體驗也會更佳.

最後,本文謹涉及基礎的ARMv8與部分軟體概念,對有志於進一步探究ARMv8的初學者希望能有所助益.

December 16th, 2014 | Category: Loda的技術文章分享