GDB

- GDB or GNU debugger is a standard debugger for GNU based OS like Linux.

How to start gdb?

- We compile our program like this

```
cc -g -o program program.c
gdb program
```

The g option helps us to know where the program is failing.

- Now we enter gdb program
- How our gdb program is running over our program named "program".
- To run a program now with gdb debugger we enter the command run.
- Whenever the program faults the gdb gives the location and reason for failure.

Stacktrace and Examining Variables

- To back trace the call flow of program to get to know from where the problem would have originated in program execution we enter backtrace command while running gdb.
- We can also know the values of variables in current state when the program had faulted by using print command, eg. print i; this will give us the value of variable i when the program had faulted.
- gdb keeps these results in a pseudo variable starting with $1, $2 and so.
- To print a number of consecutive items we use print array[0]@n , where n is the number of items that we want to print and array is the array name.

Listing the program and Setting Breakpoints

- When we are within gdb and want to view the source code of the program then we use list command, this will print the few lines before and few lines after the program faulted while running.
- To display more lines we use list command again.
- To set a breakpoint at a particular line we enter these set of commands

```
gdb program
break lineno
run
```

- The program stops where we have set the breakpoint and prints the value of variable at that point belonging to that line.
- To continue the program we use cont command.
- We can use display command to set gdb to display the values of variables whenever the program stops at a breakpoint eg. display k; display array[0]@4.

- To come out of gdb anytime we use quit command.
- We can see the breakpoints and displays we have enabled using these commands

info break

info display

- To disable breakpoint and display we use

disable break # (# is the breakpoint number we see using info break command).

disable display # (# is the display we have set using info display command).

What is a Loadable Kernel Module?

Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. Example: one type of module is the device driver, which allows the kernel to access hardware connected to the system.
Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality.
C

- kernel modules are object files that contain kernel code.
- They can be loaded and removed from the kernel during run time.
- They contain unresolved symbols that are linked into the kernel when the module is loaded.
- kernel modules can only do some of the things that built-in code can do , they do not have access to internal kernel symbols.dep

Kernel Module Utilities:

lsmod : lists the modules already loaded into kernel.

rmmod : Removes or unloads a module one at a time.

insmod : Insert or load a module.

depmod : Creates the data base of module dependencies. This is created based on the information present in /lib/modules/module.dep file.

modprob : Inserts a module and its dependencies based on information from modules.depfile.

modinfo : List the information about the module like author, version tag, parameters etc.

Writing a simple kernel module:
Here I am explaining how to write a simple kernel module that doesn't have any functionality.

Every Kernel modules must have at least two functions:
"start" (initialization) function called init_module()
It is called when the module is loaded using insmod into the kernel.

"end" (cleanup) function called cleanup_module()
It is called just before it is removed using rmmod.

Kmod is a subsystem that allows the loading and unloading of modules.

Issues to be considered in writing a kernel module:

- Module code should not invoke user space Libraries or API's or System calls.
- Modules are free to use kernel data types and GNU-C extensions for linux kernel.
- Following path contains the list of header files that can be included in module programs./lib/modules/2.6.32.generic/build/include/linux.

Code for Test.c:

```
#include <linux/module.h>  /* Needed by all modules */
#include <linux/kernel.h>/* Needed for KERN_INFO */
#include <linux/init.h>
#include <linux/version.h>

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("This is a basic test module);
MODULE_AUTHOR("xyz");

int init_module(void)
{
printk(KERN_INFO "My first test module loaded.\n");
return 0;
}
void cleanup_module(void)
{
printk(KERN_INFO "Test module unloaded\n");
}
```

Note: Copying the code directly into your source.c file will also copies the invisible characters and finally you will left out with stray errors, i recommend you to type the code and that even becomes a practice.

module.h : module management subsystem, its an interface file for Kmod functions.
kernel.h : resolves kernel symbol calls, it provides access to global symbol table.
init.h : describes the sequence of initialization.
version.h : It binds a module to a particular version of kernel.
printk : printk is printf for kernel programs, as said earlier modules can't use stdlib due to user space/ kernel space issues. Most of C library is implemented in kernel. with in printk "KERN_INFO" is a macro found in kernel.h that defines the priority to printk logs. There are 8 such macros as shown below.

0- highest priority 7-lowest priority

KERN_EMERG "<0>" /* system is unusable */
KERN_ALERT "<1>" /* action must be taken immediately */
KERN_CRIT "<2>" /* critical conditions */
KERN_ERR "<3>" /* error conditions */
KERN_WARNING "<4>" /* warning conditions */
KERN_NOTICE "<5>" /* normal but significant condition */
KERN_INFO "<6>" /* informational */
KERN_DEBUG "<7>" /* debug-level messages */

MACROS :
MODULE_LICENSE() : declares the module's licensing
MODULE_DESCRIPTION() : to describe what the module does
MODULE_AUTHOR() : declares the module's author

- In modules the comments are achieved with macros so that information of module sits along with the code so that debugging becomes easy. i.e., comments are not stripped out.
- License macro is mandatory even if it is free or proprietary.
- Modules can comprise of any number of functions and data elements which form module body.

Building a Module:
Kernel source has two types of make files:

1. src/Makefile called as top/primary Makefile
2. Each branch in kernel source has a Makefile called as kbuild Makefile.

So to build a module we have to write a Makefile that follows the rules followed by kbuild. To learn more on how to compile modules, see file linux/Documentation/kbuild/modules.txt.

Makefile:

```
obj-m += test.o

all:
 make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
 make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

from a technical point of view the first line is really necessary, the "all" and "clean" targets were added for convenience, and make sure that after "all:" give a tab and write the command as its a convention in writing Makefile.
Now you can compile the module test.c using make command. Here is the list of command I am executing the corresponding output is show in below image highlighted with blue arrow.
Note: You should be the root user or should have requisite permissions to load and unload modules.

```
$ ls    (to list the files in the current directory)
$ make      ( to build the module, this will generate many files)
$ ls      (to list the files in the current directory)
$ insmod test.ko      ( loading the module)
$ dmesg     ( to see the messages printed by printk)
```

$ modinfo test.ko (this display the module information, we have put in macros)
$ rmmod test (unloading the module)
$ dmesg (to see the messages printed by printk)

What are monolithic and micro kernels and what are the differences between them?

Monolithic kernel is a single large processes running entirely in a single address space. It is a single static binary file. All kernel services exist and execute in kernel address space. The kernel can invoke functions directly. The examples of monolithic kernel based OSs are Linux, Unix.

In Microkernels, the kernel is broken down into separate processes, known as servers. Some of the servers run in kernel space and some run in user-space. All servers are kept separate and run in different address spaces.The communication in microkernels is done via message passing. The servers communicate through IPC (Interprocess Communication). Servers invoke "services" from each other by sending messages. The separation has advantage that if one server fails other server can still work efficiently. The example of microkernel based OS are Mac OS X and Windows NT.

Differences--

1) Monolithic kernel is much older than Microkernel. It’s used in Unix . While Idea of microkernel appeared at the end of the 1980's.
2) the example of os having the Monolithic kernels are UNIX , LINUX .While the os having Microkernel are QNX , L4 , HURD , initially Mach (not mac os x) later it will converted into hybrid kernel , even MINIXis not pure kernel because device driver are compiled as part of the kernel .
3) Monolithic kernel are faster than microkernel . While The first microkernel Mach is 50% slower than Monolithic kernel while later version like L4 only 2% or 4% slower than the Monolithic kernel .
4) Monolithic kernel generally are bulky . While Pure monolithic kernel has to be small in size even fit in s into processor first level cache (first generation microkernel).
5) In the Monolithic kernel device driver reside in the kernel space . While In the Microkernel device driver reside in the user space.
6) Since the device driver reside in the kernel space it make monolithic kernel less secure than microkernel . (Failure in the driver may lead to crash) While Microkernels are more secure than the monolithic kernel hence used in some military devices.
7) Monolithic kernels use signals and sockets to ensure IPC while microkernel approach uses message queues . 1 gen of microkernel poorly implemented IPC so were slow on context switches.
8) Adding new feature to a monolithic system means recompiling the whole kernel While You can add new feature or patches without recompiling.

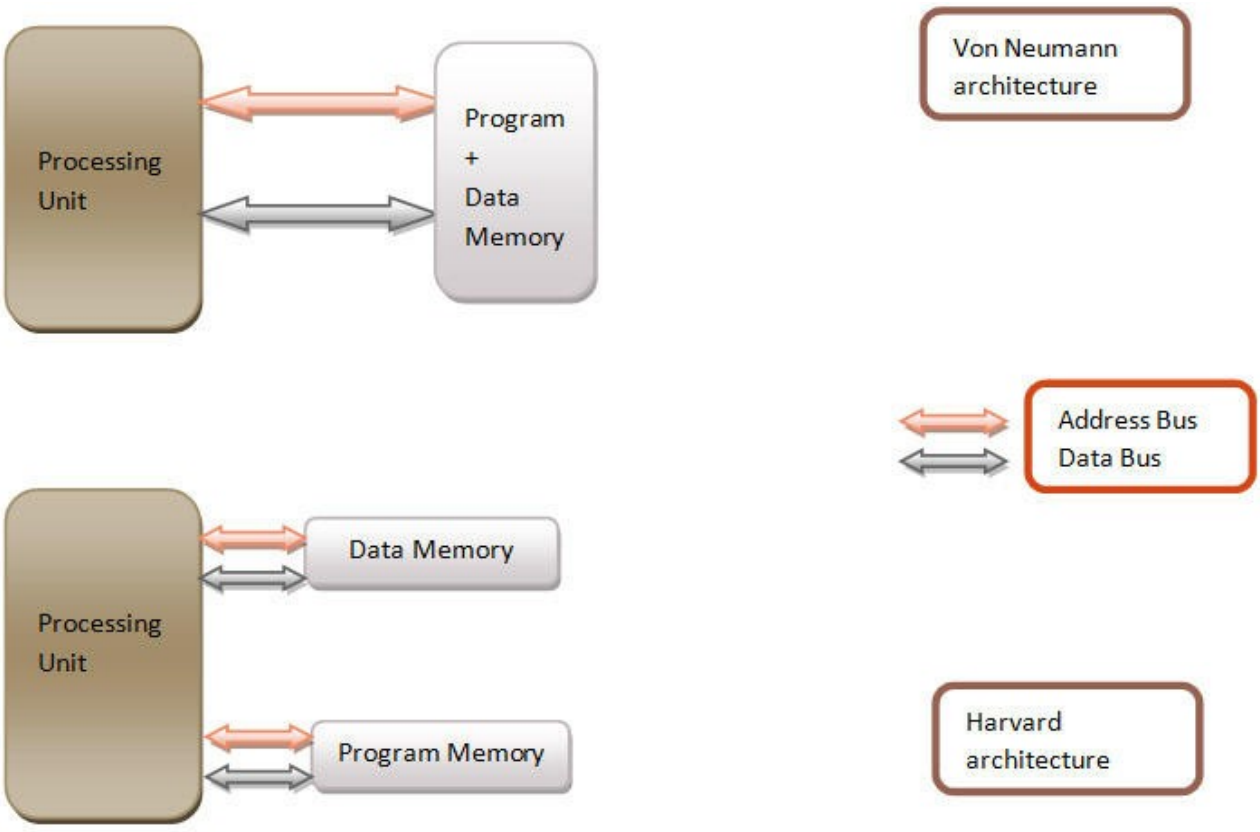
ARM Architecture

- It is a spinoff from a UK based company Acorn Computers.

- Its name has changes from Acorn RISC Machine to Advanced RISC Machine and now simply ARM.
- ARM doesnt produce Silicon, it just provides RISC processor core designs(Physical IPs).
- There are two category of ARM processors- Embedded Cortex Processor, Application Cortex Processors.
- Embedded Cortex Processors consist of -Microprocessors and Real Time Processors.
- Application Cortex Processor consists of 32 bit ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, ARM Cortex-A17 MPCore. They need an platform OS(linux), has extended instruction set and a good memory management technique.Newer cortex A series processors supports multicore.
- he ARM Cortex-M is a group of 32-bit RISC ARM processor cores licensed by ARM Holdings. The cores are intended for microcontroller use, and consist of the Cortex-M0, Cortex-M0+, Cortex-M1, Cortex-M3, and Cortex-M4 .
- The ARM Cortex-R is a group of 32-bit RISC ARM processor cores licensed by ARM Holdings. The cores are intended for robust real-time use, and consists of the Cortex-R4, Cortex-R5, Cortex-R7.

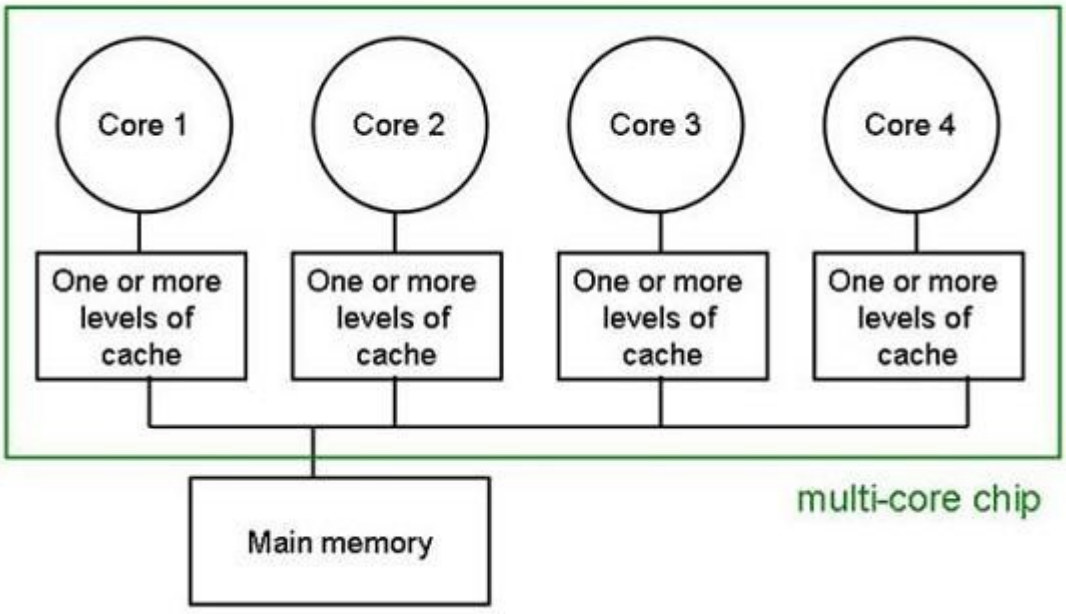
Von Neuman and Harvard Architecture

Before we move ahead we must look at these two architectures as some ARM architecture use one and some the other one.

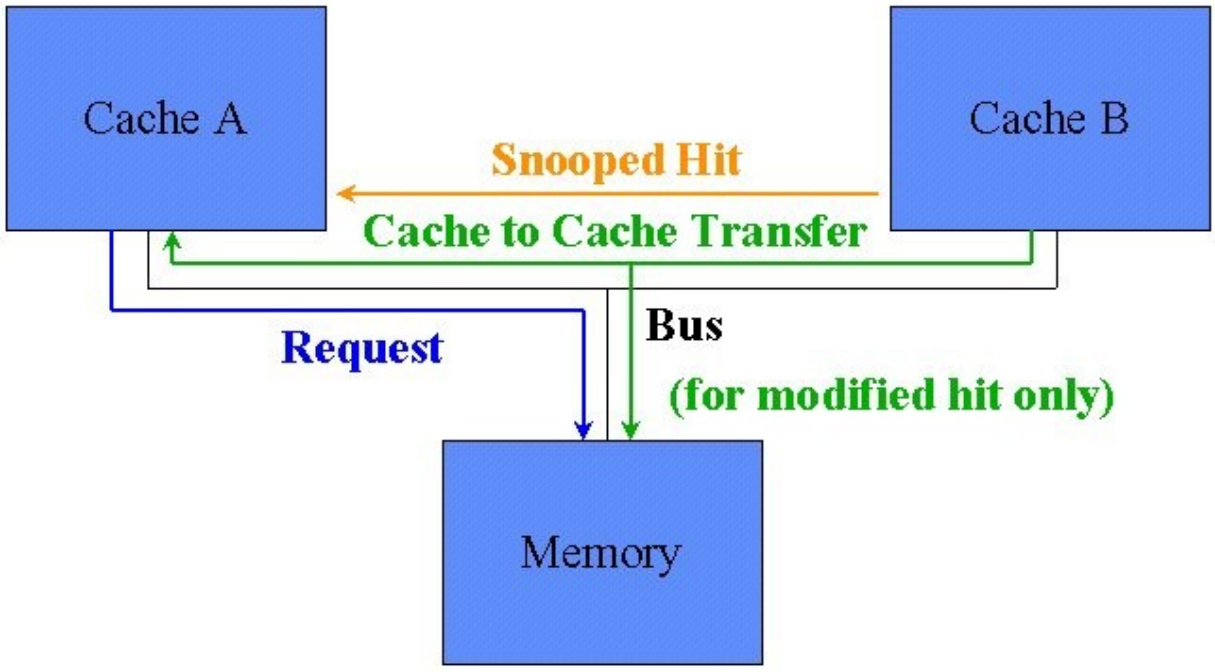


In Von Neuman Architecture the Instruction(Program) and Data both reside in the same memory segment whereas in Harvard Architecture both reside separately. In Von Neuman Architecture we can get either instruction or data to the cache(which will ultimately goto the ALU) in one clock cycle whereas in Harvard Architecture we can get both the instruction and data to the respective caches in the same clock cycle itself. This is achieved because in the idle time the address and data bus keeps on filling the caches so that when t he processing unit requests for it , it can be given. ARM v7 uses Von Neuman Architecture whereas ARM 9 uses Harvard Architecture.

Snooping



Snooping is a cache coherency protocol used in ARM systems with multiple cores and thus multiple caches. I will first explain the problem that can occur because of this setup. Suppose the same data , say X from the main memory is there in two caches(of different cores say core 1 and core 4). Suppose the core 1 performs some operation on data X and it gets modified to Y. But the cache of core 4 will still have data X i.e it doesn't have updated data. So, if the core 4 needs to perform some operation then it will do it in the old data and this is undesirable. The diagram below will show that how different caches do snooping on data being requested by other cache from the memory. If the data is being requested from the same memory location from where the other cache has data then cache hit happens and the other cache transfers the data to the cache which is requesting from the memory.



Fast Burst Access

The memory is divided into horizontal and vertical array of cells. For the addresses in same vertical line we have Row access and in different vertical line we have Column Access. If the addresses are in the same verical line then out of 32 bits of the address only 16 bits has to be changed and the other 16 bit is constant.
This makes access faster and its called fast burst mode.

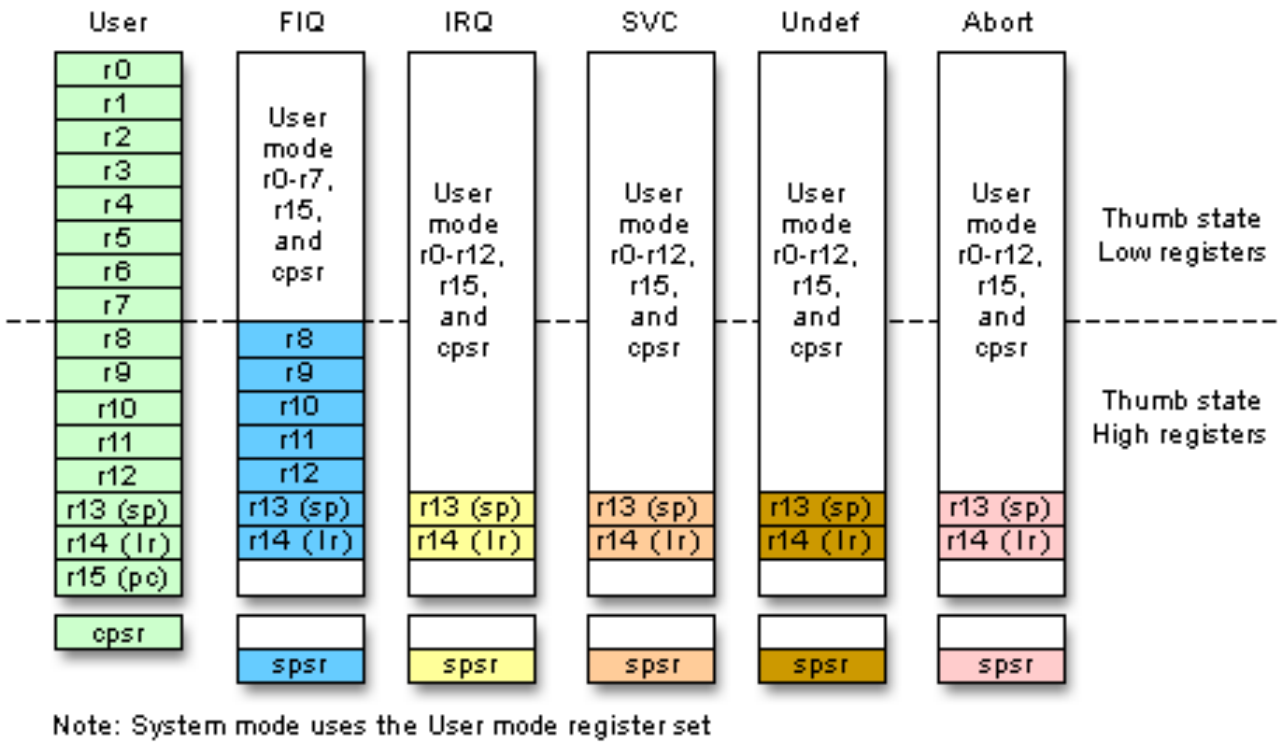
Static Memory and Dynamic Memory

Static memories are those memories which don't need to be refreshed regularly whereas dynamic memories need to be refreshed after every 14ns then only they keep data. To store the same data static memory uses 10 times more transistors as compared to dynamic memories.

ARM 7 Architecture

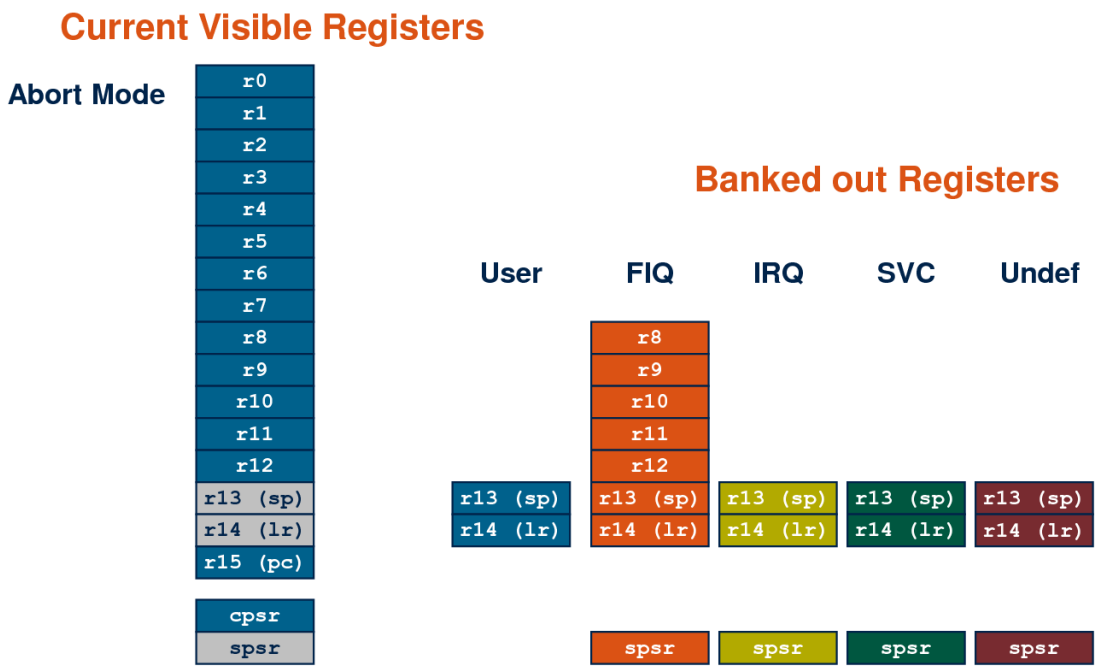
It has 37 registers (31 normal registers and 6 status registers). Barrel shifter can multiply the second operand by shifting the bits. It has 32 8 bit multipliers also for multiplication operations.

ARM Register Sets



There are 37 registers in total. We can see that when the core enters abort mode from user mode then the registers of abort mode are swapped in and those of user mode are swapped out. The contents of cpsr is copied into spsr and when the mode is exited then the contents of sprs will be copied into cpsr.

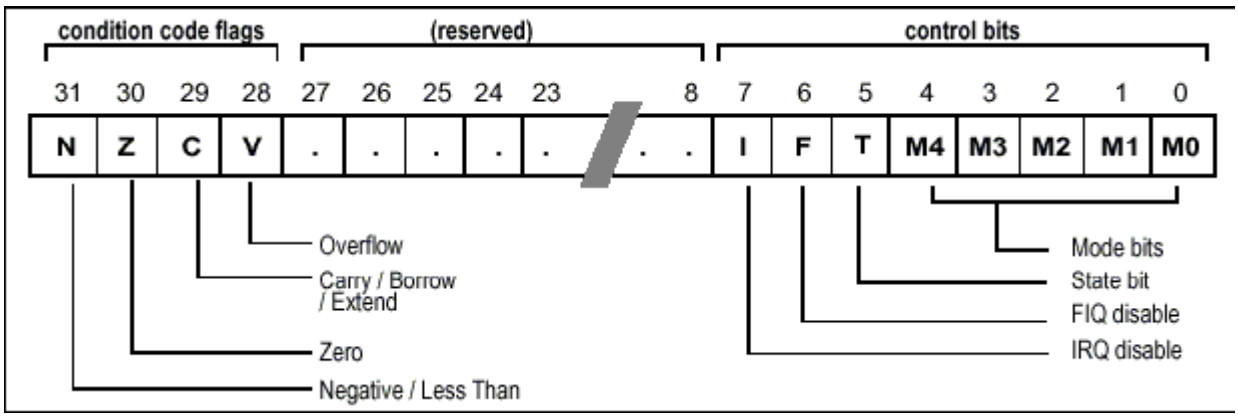
The ARM Register Set



Explaination of Modes

	Mode	Description	
Exception modes	Supervisor (SVC)	Entered on reset and when a Supervisor call instruction (SVC) is executed	Privileged modes
	FIQ	Entered when a high priority (fast) interrupt is raised	
	IRQ	Entered when a normal priority interrupt is raised	
	Abort	Used to handle memory access violations	
	Undef	Used to handle undefined instructions	
	System	Privileged mode using the same registers as User mode	
	User	Mode under which most Applications / OS tasks run	Unprivileged mode

FIQ uses separate registers as it is used to handle very fast interrupts(very high priority) so we want the current registers to not get disturbed so it uses separate set of registers.



PSR(program status register) CPSR and SPSR

M[4:0]	Mode	Accessible register set	
10000	User	PC, R14..R0	CPSR
10001	FIQ	PC, R14_fiq..R8_fiq, R7..R0	CPSR, SPSR_fiq
10010	IRQ	PC, R14_irq..R13_irq, R12..R0	CPSR, SPSR_irq
10011	Supervisor	PC, R14_svc..R8_svc, R7..R0	CPSR, SPSR_svc
10111	Abort	PC, R14_abt..R8_abt, R7..R0	CPSR, SPSR_abt
11011	Undefined	PC, R14_und..R8_und, R7..R0	CPSR, SPSR_und

Jezelle

Jazelle DBX (Direct Bytecode eXecution) is a technique that allows Java Bytecode to be executed directly in the ARM architecture as a third execution state (and instruction set) alongside the existing ARM and Thumb-mode. Support for this state is signified by the "J" in the ARMv5TEJ architecture, and in ARM9EJ-S and ARM7EJ-S core names. Support for this state is required starting in ARMv6 (except for the ARMv7-M profile), though newer cores only include a trivial implementation that provides no hardware acceleration.

Thumb[edit]

Thumb and Thumb2 Instructions

o improve compiled code-density, processors since the ARM7TDMI (released in 1994) have featured the Thumb instruction set, which have their own state. (The "T" in "TDMI"

indicates the Thumb feature.) When in this state, the processor executes the Thumb instruction set, a compact 16-bit encoding for a subset of the ARM instruction set.Most of the Thumb instructions are directly mapped to normal ARM instructions. The space-saving comes from making some of the instruction operands implicit and limiting the number of possibilities compared to the ARM instructions executed in the ARM instruction set state.

In Thumb, the 16-bit opcodes have less functionality. For example, only branches can be conditional, and many opcodes are restricted to accessing only half of all of the CPU's general-purpose registers. The shorter opcodes give improved code density overall, even though some operations require extra instructions. In situations where the memory port or bus width is constrained to less than 32 bits, the shorter Thumb opcodes allow increased performance compared with 32-bit ARM code, as less program code may need to be loaded into the processor over the constrained memory bandwidth.

Embedded hardware, such as the Game Boy Advance, typically have a small amount of RAM accessible with a full 32-bit datapath; the majority is accessed via a 16-bit or narrower secondary datapath. In this situation, it usually makes sense to compile Thumb code and hand-optimise a few of the most CPU-intensive sections using full 32-bit ARM instructions, placing these wider instructions into the 32-bit bus accessible memory.

Thumb-2 technology was introduced in the ARM1156 core, announced in 2003. Thumb-2 extends the limited 16-bit instruction set of Thumb with additional 32-bit instructions to give the instruction set more breadth, thus producing a variable-length instruction set. A stated aim for Thumb-2 was to achieve code density similar to Thumb with performance similar to the ARM instruction set on 32-bit memory.

Why do we need two bootloaders viz. primary and secondary?

When the system starts the BootROM has no idea about the external RAM. It can only access the Internal RAM of the the CPU. So the BootROM loads the primary bootloader from the boot media (flash memory) into the internal RAM. The main job of the primary bootloader is to detect the external RAM and load the secondary bootloader into it. After this, the secondary bootloader starts its execution.

Given a pid, how will you distinguish if it is a process or a thread ?

Do ps -AL | grep pid
1st column is parent id and the second column is thread (LWP) id. if both are same then its a process id otherwise thread.

What are the Synchronization techniques used in Linux Kernel?

For simple counter variables or for bitwise ------->atomic operations are best methods.

atomic_t count=ATOMIC_INIT(0); or atomic_set(&count,0);
atomic_read(&count);
atomic_inc(&count);
atomic_dec(&count);
atomic_add(&count,10);
atomic_sub(&count,10);

Spinlocks are used to hold critical section for short time and can use from interrupt context and locks can not sleep,also called busy wait loops.
fully spinlocks and reader/writer spin locks are available.
spinlock_t my_spinlock;
spin_lock_init(&my_spinlock);
spin_lock(&my_spinlock);
// critical section
spin_unlock(&my_spinlock);
Spinlock variant with local CPU interrupt disable
spin_lock_irqsave(&my_spinlock, flags);
// critical section
spin_unlock_irqrestore(&my_spinlock, flags);
if your kernel thread shares data with a bottom half,
spin_lock_bh(&my_spinlock);
// critical section
spin_unlock_bh(&my_spinlock);
If we have more readers than writers for our shared resource
Reader/writer spinlock can be used

rwlock_t my_rwlock;
rwlock_init(&my_rwlock);
write_lock(&my_rwlock);
// critical section -- can read and write
write_unlock(&my_rwlock);

read_lock(&my_rwlock);
// critical section -- can read only
read_unlock(&my_rwlock);

Mutexs are used when we hold lock for longer time and if we use from process context.
DEFINE_MUTEX(my_mutex);
mutex_lock(&my_mutex);
mutex_unlock(&my_mutex);

How does Linux Manage memory?

1> Bring pages in only when needed (demand paging of text sections)
2> Keep pages brought into memory as long as possible to avoid reading from slower devices (page cache etc.)

3> Under memory pressure, get rid of redundant copies of data (text pages being freed during memory pressure)
4> If required memory can not be freed as there is only one copy of it (data sections), move the data to slower medium(swapping onto disk) and book keep the information so that they can be brought back into memory when needed.
5> Use processor MMUs for isolation and privileges. These MMU components (TLBs and caches) act as a cache of page tables.
6> Use optimized algorithms and data structures.

Explain the module loading in Linux.

A module can be loaded to Linux Kernel in two ways
1. Statically
2. Dynamically

Static loading means that the module is loaded in the memory with the kernel loading itself.

Dynamic loading means that the module is loaded into the kernel at the run time.
The command that is used to achieve it is insmod.

The user must have the root permission to do so.
e. sudo insmod test.ko

When should one use Polling and when should one use Interrupts?

Both the mechanisms have their own pluses and minuses.

We should use interrupts when we know that our event of interest is-

1. Asynchronous
2. Important(urgent).
3. Less frequent

We should use polling when we know that our event of interest is-

1. Synchronous
2. Not so important
3. Frequent(our polling routine should encounter events more than not)

What is the difference between IRQ and FIQ in case of ARM?

ARM treats FIQ(Fast interrupt request) at a higher priority as compared to IRQ(interrupt request). When an IRQ is executing an FIQ can interrupt it while vice versa is not true.

ARM uses a dedicated banked out register for FIQ mode ; register numbers R8-R12.
http://learnlinuxconcepts.blogspot.in/2014/06/arm-architecture.html
So when an FIQ comes these registers are directly swapped with the normal working register and the CPU need not take the pain of storing these registers in the stack. So it makes it faster.

One more point worth noting is that the FIQ is located at the end of exception vector table(0X1c) which means that the code can run directly from 0X1C and this saves few cycles on entry to the ISR.

Where are macros stored in the memory?

#define func() func1(){...}

Macros aren't stored anywhere separately. They get replaced by the code even before compilation.The compiler is unaware of the presence of any macro. If the code that replaces macro is large then the program size will increase considerably due to repetition.

What is a kernel Panic?

It is an action taken by linux kernel when it experiences a situation from where it can't recover safely.In many cases the system may keep on running but due to security risk by fearing security breach the kernel reboots or instructs to be rebooted manually.

It can be caused due to various reasons-
1. Hardware failure
2. Software bus in the OS.
3. During booting the kernel panic can happen due to one of the reasons-
a.Kernel not correctly configured, compiled or installed.
b.Incompatibility of OS, hardware failure including RAM.
c.Missing device driver
d.Kernel unable to locate root file system
e.After booting if init process dies.

In windows operating systems the equivalent term is stop error(or, colloquiaally Blue screen of death).

What is a device tree in Linux?

Device tree is a data structure that describes the hardware , their layout, their parameters and other details and is passed to the kernel at boot time

Linux Memory Management

PAGES:

- The physical pages are the basic unit of memory management for the Kernel.
- The MMU(memory management unit) manages the memory in terms of page sizes.
- Generally a 32 bit architecture has 4KB page size and a 64 bit architecture has 8KB page size.
- Kernel stores info about these pages(physical pages) in its structure struct page.
- This structure is defined in <linux/mm.h>.

```
struct page {
    page_flags_t        flags;
```

```
    atomic_t            _count;
    atomic_t            _mapcount;
    unsigned long       private;
    struct address_space  *mapping;
    pgoff_t             index;
    struct list_head    lru;
    void                *virtual;
};
```

- Some important fields are-

1. The flags field stores the status of the page. Such flags include whether the page is dirty(it has been modified) or whether it is locked in memory. There are 32 different flags available. The flag values are defined in <linux/page-flags.h>.
2. _count field means how many instance virtual pages are there for the given physical page. When the value of _count reaches zero that means noone is using the page at current.
3. virtual this is the address of the page in virtual memory. For highmem(highmemory >896MB of virtual memory) this field is zero.

- The goal of this data structure is to describe the physical pages and not the data contained in that page.

ZONES:

- The kernel divides its 1GB virtual address space into three zones -ZONE_DMA(<16MB), ZONE_NORMAL(16MB-896MB) and ZONE_HIGHMEM(>896MB).
- Kernel groups pages with similar properties into separate zones.
- The zones have no physical relevance, it has just logical relevance.
- Each zone is represented by struct zone, which is defined in <linux/mmzone.h>:
- For more details on ZONES , read my other post on linux addressing.

GETTING PAGES:

- Kernel allows us with some interfaces to allocate and free memory within kernel space.
- All these interfaces allocate memory with page-sized granularity and are declared in <linux/gfp.h>.
- We can either allocate physical contiguous memory or only virtual contiguous memory.
- One should never attempt to allocate memory for userspace from the kernel - this is a huge violation of the kernel's abstraction layering.
- Instead have userspace mmap pages owned by your driver directly into its address space or have userspace ask how much space it needs. Userspace allocates, then grabs the memory from the kernel.
- There no way to allocate contiguous physical memory from userspace in linux.

- This is because a user space program has no way of controlling or even knowing if the underlying memory is contiguous or not.
- The core function is struct page * alloc_pages(unsigned int gfp_mask, unsigned int order)
- This allocates 2order (that is, 1 << order) contiguous physical pages and returns a pointer to the first page's page structure;on error it returns NULL.
- To convert a given page(physical) to its logical address we can use the function-void * page_address(struct page *page).
- This function returns a pointer to the logical address where our allocated physical pages resides.
- If we just need the virtual address of the pages(we don't need page structure) we can use the function -unsigned long __get_free_pages(unsigned int gfp_mask, unsigned int order).
- The pages thus obtained are contiguous in virtual space.
- This function also uses the core function alloc_pages, but it directly gives us the starting address of the first page.
- If we just need a single page(order 0 then we have two functions, one for physical and other for logical-

1. struct page * alloc_page(unsigned int gfp_mask)
2. unsigned long __get_free_page(unsigned int gfp_mask)

- If we need page filled with zero(for security issues we want to initialize memory with all zeros so that if we need to pass this memory to user space then the user space will get access to the contents written on this memory location previously) we can use this function unsigned long get_zeroed_page(unsigned int gfp_mask)
- This function works the same as __get_free_page(), except that the allocated page is then zero-filled
- To free the pages we have some functions-

1. void __free_pages(struct page *page, unsigned int order)
2. void free_pages(unsigned long addr, unsigned int order)
3. void free_page(unsigned long addr)

- Allocation of page/s may fail so we must define a handler to handle such situations.

kmalloc()

- The kmalloc() function's operation is very similar to that of user-space's familiar malloc() routine, with the exception of the addition of a flags parameter.
- This is used when we want to allocate a small chunk of memory in bytes size.
- For bigger sized memory, the previous page allocation functions is a good option.
- Mostly in Kernel we use Kmalloc() for memory allocation.
- The function is declared in <linux/slab.h>
- void * kmalloc(size_t size, int flags)
- The function returns a pointer to a region of memory that is at least size bytes in length.
- The region of memory allocated is physically contiguous.

- On error, it returns NULL.
- Kernel allocations almost always succeed, unless there is an insufficient amount of memory available.
- Still we must check for NULL after all calls to kmalloc() and handle the error appropriately.
- eg. struct abc *ptr;

  ```
  ptr = kmalloc(sizeof(struct abc), GFP_KERNEL);
  if (!ptr)
        /* handle error ... */
  ```

- The GFP_KERNEL flag specifies the behavior of the memory allocator while trying to obtain the memory to return to the caller of kmalloc().

gfp_mask Flags

In this section we will discuss about the flags that we used in kmalloc and other low level page functions.

The flags are broken up into three categories:

1. action modifiers
2. zone modifiers
3. types.

- Action modifiers specify how the kernel is supposed to allocate the requested memory.
- In certain situations, only certain methods can be employed to allocate memory.
- For example, interrupt handlers must instruct the kernel not to sleep (because interrupt handlers cannot reschedule) in the course of allocating memory.
- Zone modifiers specify from where to allocate memory.
- As we saw in the article on linux addressing (http://learnlinuxconcepts.blogspot.in/2014/02/linux-addressing.html) the kernel divides physical memory into multiple zones, each of which serves a different purpose.
- Zone modifiers specify from which of these zones to allocate.
- Type flags specify a combination of action and zone modifiers as needed by a certain type of memory allocation.

- Type flags simplify specifying numerous modifiers; instead, we generally specify just one type flag.
- All the flags are declared in <linux/gfp.h>.
- The file <linux/slab.h> includes this header, however, so we don't often need not include it directly.

Action modifiers-

Flag	Description
__GFP_WAIT	The allocator can sleep.
__GFP_HIGH	The allocator can access emergency pools.
__GFP_IO	The allocator can start disk I/O.
__GFP_FS	The allocator can start filesystem I/O.
__GFP_COLD	The allocator should use cache cold pages.
__GFP_NOWARN	The allocator will not print failure warnings.
__GFP_REPEAT	The allocator will repeat the allocation if it fails, but the allocation can potentially fail.
__GFP_NOFAIL	The allocator will indefinitely repeat the allocation. The allocation cannot fail.
__GFP_NORETRY	The allocator will never retry if the allocation fails.
__GFP_NO_GROW	Used internally by the slab layer.
__GFP_COMP	Add compound page metadata. Used internally by the hugetlb code.

- These allocations can be specified together. For example,ptr = kmalloc(size, __GFP_WAIT | __GFP_IO | __GFP_FS);
- Lets see how this allocation will work--
- It will instruct the page allocator (function finally comes to alloc_pages() as we had seen before) that the allocation can-

1. block
2. perform I/O
3. perform filesystem operations, if needed.

- This allows the kernel great freedom in how it can find the free memory to satisfy the allocation.

Zone Modifier-

- Zone modifiers specify from which memory zone the allocation should originate.
- Normally, allocations can be fulfilled from any zone.
- The kernel prefers ZONE_NORMAL, however, to ensure that the other zones have free pages when they are needed.
- There are only two zone modifiers because there are only two zones other than ZONE_NORMAL (which is where, by default, allocations originate).

Flag	Description
__GFP_DMA	Allocate only from ZONE_DMA
__GFP_HIGHMEM	Allocate from ZONE_HIGHMEM or ZONE_NORMAL

- If none of the flags are specified, the kernel fulfills the allocation from either ZONE_DMA or ZONE_NORMAL, with a strong preference to satisfy the allocation from ZONE_NORMAL.
- We cannot specify __GFP_HIGHMEM to either __get_free_pages() or kmalloc() because these both return a logical address, and not a page structure.
- Though it is possible that these functions would allocate memory that is not currently mapped in the kernel's virtual address space and, thus, does not have a logical address.
- Only alloc_pages() can allocate high memory.
- For majority of our allocations, however, we don't need to specify a zone modifier because ZONE_NORMAL is sufficient.

Type Flags-

- The type flags specify the required action and zone modifiers to fulfill a particular type of transaction.
- Therefore, there is a good news that kernel code tends to use the correct type flag and not specify the various number of flags it would want to define.

GFP_ATOMIC	The allocation is high priority and must not sleep. This is the flag to use in interrupt handlers, in bottom halves, while holding a spinlock, and in other situations where we cannot sleep.
GFP_NOIO	This allocation can block, but must not initiate disk I/O. This is the flag to use in block I/O code when we cannot cause more disk I/O, which might lead to some unpleasant recursion.
GFP_NOFS	This allocation can block and can initiate disk I/O, if it must, but will not initiate a filesystem operation. This is the flag to use in filesystem code when we cannot start another filesystem operation.
GFP_KERNEL	This is a normal allocation and might block. This is the flag to use in process context code when it is safe to sleep. The kernel will do whatever it has to in order to obtain the memory requested by the caller. This flag should be our first choice.
GFP_USER	This is a normal allocation and might block. This flag is used to allocate memory for user-space processes.
GFP_HIGHUSER	This is an allocation from ZONE_HIGHMEM and might block. This flag is used to allocate memory for user-space processes.
GFP_DMA	This is an allocation from ZONE_DMA. Device drivers that need DMA-able memory use this flag, usually in combination with one of the above.

What all action modifier files are internally involved in Type Flags ?

GFP_ATOMIC	__GFP_HIGH
GFP_NOIO	__GFP_WAIT
GFP_NOFS	(__GFP_WAIT | __GFP_IO)
GFP_KERNEL	(__GFP_WAIT | __GFP_IO | __GFP_FS)
GFP_USER	(__GFP_WAIT | __GFP_IO | __GFP_FS)
GFP_HIGHUSER	(__GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HIGHMEM)
GFP_DMA	__GFP_DMA

Lets try to understand important Type flags.

GFP_KERNEL flag-

- The vast majority of allocations in the kernel use the GFP_KERNEL flag.
- The resulting allocation can sleep as it is normal priority allocation.
- Because the call can block, this flag can be used only from process context that can safely reschedule (that is, no locks are held and so on).
- Because this flag does not make any stipulations as to how the kernel may obtain the requested memory, the memory allocation has a high probability of succeeding.

GFP_ATOMIC flag-

- The GFP_ATOMIC flag is at the extreme end as compared to GFP_KERNEL flag.
- This flag specifies a memory allocation that cannot sleep, the allocation is very restrictive in the memory it can obtain for the caller.
- If no sufficiently sized contiguous chunk of memory is available, the kernel is not very likely to free memory because it cannot put the caller to sleep.
- Conversely, the GFP_KERNEL allocation can put the caller to sleep to swap inactive pages to disk, flush dirty pages to disk, and so on.
- Because GFP_ATOMIC is unable to perform any of these actions, it has less of a chance of succeeding (at least when memory is low) compared to GFP_KERNEL allocations
- Still the GFP_ATOMIC flag is the only option when the current code is unable to sleep, such as with interrupt handlers, softirqs, and tasklets.

GFP_NOIO and GFP_NOFS flags-

- In between these two flags are GFP_NOIO and GFP_NOFS.
- Allocations initiated with these flags might block, but they refrain from performing certain other operations.
- A GFP_NOIO allocation does not initiate any disk I/O whatsoever to fulfill the request
- On the other hand, GFP_NOFS might initiate disk I/O, but does not initiate filesystem I/O.
- One question that immediately comes to our mind. Why might you need these flags?
- They are needed for certain low-level block I/O or filesystem code, respectively
- Imagine if a common path in the filesystem code allocated memory without the GFP_NOFS flag. The allocation could result in more filesystem

operations, which would then beget other allocations and, thus, more filesystem operations! This could continue indefinitely.

- Code such as this that invokes the allocator must ensure that the allocator also does not execute it, or else the allocation can create a deadlock.
- Not surprisingly, the kernel uses these two flags only in few places.

GFP_DMA flag-

- The GFP_DMA flag is used to specify that the allocator must satisfy the request from ZONE_DMA.
- This flag is used by device drivers, which need DMA-able memory for their devices. Normally, we combine this flag with the GFP_ATOMIC or GFP_KERNEL flag

Which flag to use when??

Situation	Solution
Process context, can sleep	Use GFP_KERNEL
Process context, cannot sleep	Use GFP_ATOMIC, or perform your allocations with GFP_KERNEL at an earlier or later point when you can sleep
Interrupt handler	Use GFP_ATOMIC
Softirq	Use GFP_ATOMIC
Tasklet	Use GFP_ATOMIC
Need DMA-able memory, can sleep	Use (GFP_DMA | GFP_KERNEL)
Need DMA-able memory, cannot sleep	Use (GFP_DMA | GFP_ATOMIC), or perform your allocation at an earlier point when you can sleep

kfree()

- kfree undoes the work done by kmalloc().
- This function is declared in <linux/slab.h>.
- void kfree(const void *ptr).
- use it only for those blocks of memory that was previously allocated using kmalloc().
- eg. char *buf;
  buffer = kmalloc(BUF_SIZE, GFP_ATOMIC);
  if (!buffer)

```
/* error allocting memory ! */
```

Later, when we no longer need the memory, we must free it.
kfree(buffer);

vmalloc()

- This Kernel function is similar to user space function malloc().
- Both vmalloc() and malloc() returns virtually contiguous memory but not necessarily physically contiguous.
- In kernel we normally use kmalloc() and seldom use vmalloc().
- vmalloc is used when the requested memory size is quite big as it may not be possible to allocate a large block of contiguous memory via kmalloc() and it may fail.
- The vmalloc() function is declared in <linux/vmalloc.h> and defined in mm/vmalloc.c.
- Usage is identical to user-space's malloc(): void * vmalloc(unsigned long size).
- Usage of vmalloc() also affects the system performance.
- To free an allocation obtained via vmalloc(), we use

  void vfree(void *addr).

Concurrency and Race Conditions

Introduction

- The problem of concurrency occurs when the system tries to do more than one thing at once.
- With the use of SMP(symmetric multiprocessing, which means more than one processor or cores access the same memory) the chances of concurrent access has increased.
- Race condition means uncontrolled access to the shared data.
- This race condition leads to memory leak because the shared data is overwritten and the first wrote data is lost.
- It is advised to avoid sharing of resource while writing a program, for eg. avoid using global variables.
- It is also not possible to run away from sharing of resource as we have limited hardware and software resources.
- So how do we deal with concurrent access?

Locking/Mutual Exclusion provides access management.

Semaphores and Mutexes

- A process in linux goes to sleep("blocks") when it has nothing to do.
- Semaphores are used when the thread holding the lock can sleep.
- Semaphore framwork consists of a single integer value plus a pair of functions (P and V).
- Lets understand a semaphore operation-

A process which wants to access a critical region calls the function P, if the semaphone value is greater than 0 then we decrement the semaphore value by one and the process continues; if the semaphore ale is less than or equal to 0, then the process waits for some other process to release the lock.

A process which wants to get out of the critical region calls the function V, increments the semaphore value by one and wakes up the processes that are sleeping.

- When we want our semaphore to be accessed by only one process at a time then such semaphore is called a Mutex(mutual exclusion).

Linux Semaphore Implementation

- To use semaphore in our file we must include the header file <asm/semaphore.h> header file.
- We create semaphore directly and then set it to some value. //void sema_init(struct semaphore *sem, int val);
- For mutex also we can do similar things-
- DECLARE_MUTEX(name); //initialises the mutex with 1
- DECLARE_MUTEX_LOCKED(name); //initialises the mutex with 0

If the mutex has to be allocated dynamically then we use either of these two functions-

void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);

In linux terminology the P function is called down and V function is called up function.

Semaphore Methods
sema_init(struct semaphore *, int) //Initializes the dynamically created semaphore to the given count

init_MUTEX(struct semaphore *) //Initializes the dynamically created semaphore with a count of one
init_MUTEX_LOCKED(struct semaphore *) //Initializes the dynamically created semaphore with a count of zero (so it is initially locked)

down_interruptible (struct semaphore *) //Tries to acquire the given semaphore and enter interruptible sleep if it is contended
down(struct semaphore *) //Tries to acquire the given semaphore and enter uninterruptible sleep if it is contended
down_trylock(struct semaphore *) //Tries to acquire the given semaphore and immediately return nonzero if it is contended
up(struct semaphore *) //Releases the given semaphore and wakes a waiting task, if any

- For different critical sections we use different semaphore names.

ReaderWriter Semaphores

- rwsems is a special type of semaphore, we need to include <linux/rwsem.h>
- It is used rarely but is useful.
- rwsem allows either one writer or an unlimited number of readers to hold the semaphore
- rwsem must be initialised with void init_rwsem(struct rw_semaphore *sem);
- For read only access

void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);

- Similary for writers we have

void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);
void downgrade_write(struct rw_semaphore *sem);// used when we want writers to finish write fast and read to take major timeslice

- By default writers are given more priority over readers.
- Thats why we use rwsemaphore when write access is used rarely and that too for a short period of time.

Completions

- This feature was added in kernel version 2.4.7
- This is a lightweight mechanism with only one task, it allows one thread to tell other one that the task is done.
- The headerfile to be used for this is <linux/completion.h>

- A completion can be created with:

DECLARE_COMPLETION(my_completion);

- Or, if we want the completion to be created and initialized dynamically:

```
struct completion my_completion;
/* ... */
init_completion(&my_completion);
```

- We wait for the completion by calling:

void wait_for_completion(struct completion *c);

- This function performs an uninterruptible wait.
- If our code calls wait_for_completion and nobody ever completes the task, the result will be an unkillable process.

The actual completion event may be signalled by calling one of the following:

void complete(struct completion *c);
void complete_all(struct completion *c);

- The two functions behave differently if more than one thread is waiting for the same completion event. complete wakes up only one of the waiting threads while complete_all allows all of them to proceed.

- If we use complete_all, we must reinitialize the completion structure before reusing it.
- The macro:INIT_COMPLETION(struct completion c); can be used to quickly perform this reinitialization.
- Lets see an example of completion variable being used.

DECLARE_COMPLETION(comp);

ssize_t complete_read (struct file *filp, char __user *buf, size_t count, loff_t

*pos)

```
{
 printk(KERN_DEBUG "process %i (%s) going to sleep\n",
 current->pid, current->comm);
 wait_for_completion(&comp);
 printk(KERN_DEBUG "awoken %i (%s)\n", current->pid, current->comm);
 return 0; /* EOF */
}
ssize_t complete_write (struct file *filp, const char __user *buf, size_t count,
 loff_t *pos)
{
 printk(KERN_DEBUG "process %i (%s) awakening the readers...\n",
 current->pid, current->comm);
 complete(&comp);
 return count; /* succeed, to avoid retrial */
}
```

- We can see that the read operation will wait if the write opeation is being performed(till it finishes).

Spinlocks

- Unlike semaphores, spinklocks can be used in the code that can't sleep eg. interrupt handlers
- Spinlock offers better performance than semaphore but there are some constraints also.
- A spinlock is a mutual exclusion device that has only 2 states; locked/unlocked.
- In coding perspective this lock sets/unsets a bit.

- If the lock is unavailable then the code goes into a tight loop where it repeatedly checks the lock until it is available.
- Care must be taken for setting and unsetting the bit as many threads are in loop that may set.
- Spinlocks are designed to be used on a multiprocessor system.
- Single processor system running a preemeptive code also behaves as an SMP system from the point of view of concurrency.
- If a nonpreemeptive code running on a processor takes a lock then it would run till eternity as no other thread would ever be able to obtain the CPU to release the lock. For this reason, spinlock operation on a uniprocessor system is designed to perform NOOP(nooperation) and at someplaces whe change the masking statuses of some IRQs.
- To use the spinlock we must include the header file s <linux/spinlock.h>
- This initialization can be done at compile time by using spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
- Or the initialization can be done at runtime with void spin_lock_init(spinlock_t *lock);
- Before entering a critical section our code must obtain the requisite lock with:

void spin_lock(spinlock_t *lock);

- Since all spinlock waits are, by their nature, uninterruptible so, once we callspin_lock, we will spin until the lock becomes available to our code .
- To release a lock that we have obtained we call void spin_unlock(spinlock_t *lock);
- Lets us consider a practical scenario

We have taken a spinlock in our driver code and while holding the lock our driver experiences a function which will change the context of code for eg. copy_to_user() or put the process to sleep or say any interrupt comes that throws our driver code out of the processor. Our driver code is still holding the lock and won't free it for some other task.that wants it.

- This situation is undesirable so we acquire spinlock where the code is atomic and it can't sleep.
- For example we dont use spinlock for malloc as it can sleep.
- The kernel code takes care of spin lock and preemption, whenever spinlock is called the kernel blocks the preemption(interrupts) on that processor
- Even in case of uniprocessor system preemption is disabled to avoid the race condition.
- We must take care that our process must holdthe spinlock for a small period of time, beacuse longer is the hold longer is the wait for other process which whishes to acquire the lock,(the process might be a high priority one).
- There are four functions that can lock a spinlock

void spin_lock(spinlock_t *lock);//Normal one

void spin_lock_irqsave(spinlock_t *lock, unsigned long flags); //disables interrupts (on the local processor only) before taking the spinlock and the previous interrupt state is stored in flags

void spin_lock_irq(spinlock_t *lock);//used when we are sure nothing else might have diabled the interrupts already, it also disables interrupt

void spin_lock_bh(spinlock_t *lock);//disables the software interrupt before taking the lock

- We must use the proper lock in proper situation- eg. hardware interrupt and software interrupt or normal case.
- Just as we saw the 4 ways to acquire the spinlock we have 4 ways to release also

void spin_unlock(spinlock_t *lock);

void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);

void spin_unlock_irq(spinlock_t *lock);

void spin_unlock_bh(spinlock_t *lock);

- There is also a set of nonblocking spinlock operations:
- int spin_trylock(spinlock_t *lock);
- int spin_trylock_bh(spinlock_t *lock);
- These functions return nonzero on success (the lockwas obtained), 0 otherwise.
- There is no “try” version that disables interrupts.

Reader/Writer Spinlocks

- This is analogous to what we had seen in case of semaphores, a single writer and any number of readers.
- rwlock_t my_rwlock = RW_LOCK_UNLOCKED; /* Static way */
- rwlock_t my_rwlock;
- rwlock_init(&my_rwlock); /* Dynamic way */
- For readers, the following functions are available:

void read_lock(rwlock_t *lock);

void read_lock_irqsave(rwlock_t *lock, unsigned long flags);

void read_lock_irq(rwlock_t *lock);

```
void read_lock_bh(rwlock_t *lock);
void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
int write_trylock(rwlock_t *lock);
void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

Alternatives to Locking ?

There are situations in kernel programming where atomic access can be set up without the need for full locking.

Atromic Operation- These operations are performed in a single machine cycle.

An atomic_t holds an int value on all supported architectures.

Bit operations-

The atomic_t type is good for performing integer arithmetic. It doesn’t workas well,

however, when you need to manipulate individual bits in an atomic manner. For that

purpose, instead, the kernel offers a set of functions that modify or test single bits

atomically. Because the whole operation happens in a single step, no interrupt (or other processor) can interfere.

Seqlocks-

Used when the resource to be protected is small, simple and frequently accessed.Here readers are allowed the free access to the resource but they check for their collision with the writers. And if a colission is detected then they retry to read.

Interrupts

INTERRUPTS

- The kernel is responsible for servicing the request of hardwares.
- The CPU must process the request from the hardware.
- Since the CPU frequency and the hardware frequency is not the same(hardware is slower) so the the hardwares can't send the data/request to the CPU synchronously.
- There are two ways in which CPU can check about the request from a hardware-

1. Polling
2. Interrupt

- In polling the CPU keeps on checking all the hardwares of the availablilty of any request.
- In interrupt the CPU takes care of the hardware only when the hardware requests for some service.

- Polling is an expensive job as it requires a grater overhead.
- The better way is to use interrupt as the hardware will request the CPU only when it has some request to be serviced.
- Different devices are given different interrupt values called IRQ (interrupt request) lines.
- For ex. IRQ zero is the timer interrupt and IRQ one is the keyboard interrupt.
- An interrupt is physically produced by electronic signals originating from hardware devices and directed into input pins on an interrupt controller.
- Some interrupt numbers are static and some interrupts are dynamically assigned.
- Be it static or dynamic, the kernel must know which interrupt number is associated with which hardware.
- The interrupt controller, in turn, sends a signal to the processor. The processor detects this signal and interrupts its current execution to handle the interrupt.
- The processor can then notify the operating system that an interrupt has occurred, and the operating system can handle the interrupt appropriately.
- Interrupt handlers in Linux need not be reentrant. When a given interrupt handler is executing, the corresponding interrupt line is masked out on all processors, preventing another interrupt on the same line from being received. Normally all other interrupts are enabled, so other interrupts are serviced, but the current line is always disabled.

Comparison between interrupts and exceptions-

- Exceptions occur synchronously with respect to the processor clock while interrupts occur async.
- That is why exceptions are often called synchronous interrupts.
- Exceptions are produced by the processor while executing instructions either in response to a programming error (for example, divide by zero) or abnormal conditions that must be handled by the kernel (for example, a page fault).
- Many processor architectures handle exceptions in a similar manner to interrupts, therefore, the kernel infrastructure for handling the two is similar.
- Exceptions are of two types- traps and software interrupts.
- Exceptions are produced by the processor while executing instructions either in response to a programming error (for example, divide by zero) or abnormal conditions that must be handled by the kernel (for example, a page fault).

- A trap is a kind of exceptions, whose main purpose is for debugging (eg. notify the debugger that an instruction has been reached) or it occurs during abnormal conditions.
- A software interrupt occur at the request of a programmer eg. System calls.

Interrupt Handler

- These are the C functions that get executed when an interrupt comes.
- Each interrupt is associated with a particular interrupt handler.
- Interrupt handler is also known as interrupt service routine (ISR).
- Since interrupts can come any time therefore interrupt handlers has to be short and quick.
- At least the interrupt handler has to acknowledge the hardware and rest of the work can be done at a later time.

Top Halves Versus Bottom Halves

- There are two goals that an interrupt handler needs to perform 1. execute quickly and 2. perform a large amount of work .
- Because of these conflicting goals, the processing of interrupts is split into two parts, or halves.
- The interrupt handler is the top half.
- It is run immediately upon receipt of the interrupt and performs only the work that is time critical, such as acknowledging receipt of the interrupt or resetting the hardware.
- Work that can be performed later is delayed until the bottom half.
- The bottom half runs in the future, at a more convenient time, with all interrupts enabled.
- Let us consider a case where we need to collect the data form a data card and then process it.
- The most important job is to collect the data from data card to the memory and free the card for incoming data and this is done in top half.
- The rest part which deals with the processing of data is done in the bottom half.

Registering an Interrupt Handler

- Interrupt handlers are the responsibility of the driver managing the hardware. Each device has one associated driver and, if that device uses interrupts (and most do), then that driver registers one interrupt handler.
- Drivers can register an interrupt handler and enable a given interrupt line for handling via the function

```
/* request_irq: allocate a given interrupt line */
int request_irq(unsigned int irq,
                irqreturn_t (*handler)(int, void *, struct pt_regs *),
                unsigned long irqflags,
                const char *devname,
                void *dev_id)
```

- The first parameter, irq, specifies the interrupt number to allocate. For some devices, for example legacy PC devices such as the system timer or keyboard, this value is typically hard-coded. For most other devices, it is probed or otherwise determined programmatically and dynamically.
- The second parameter, handler, is a function pointer to the actual interrupt handler that services this interrupt. This function is invoked whenever the operating system receives the interrupt. Note the specific prototype of the handler function: It takes three parameters and has a return value of irqreturn_t.
- The third parameter, irqflags, might be either zero or a bit mask of one or more of the following flags:

- SA_INTERRUPT This flag specifies that the given interrupt handler is a fast interrupt handler. Fast interrupt handlers run with all interrupts disabled on the local processor. This enables a fast handler to complete quickly, without possible interruption from other interrupts. By default (without this flag), all interrupts are enabled except the interrupt lines of any running handlers, which are masked out on all processors. Sans the timer interrupt, most interrupts do not want to enable this flag.
- SA_SAMPLE_RANDOM This flag specifies that interrupts generated by this device should contribute to the kernel entropy pool. The kernel entropy pool provides truly random numbers derived from various random events. If this flag is specified, the timing of interrupts from this device are fed to the pool as entropy. Do not set this if your device issues interrupts at a predictable rate (for example, the system timer) or

can be influenced by external attackers (for example, a networking device). On the other hand, most other hardware generates interrupts at nondeterministic times and is, therefore, a good source of entropy.

- SA_SHIRQ This flag specifies that the interrupt line can be shared among multiple interrupt handlers. Each handler registered on a given line must specify this flag; otherwise, only one handler can exist per line. More information on shared handlers is provided in a following section.
- The fourth parameter, devname, is an ASCII text representation of the device associated with the interrupt. For example, this value for the keyboard interrupt on a PC is "keyboard". These text names are used by /proc/irq and /proc/interrupts for communication with the user, which is discussed shortly.
- The fifth parameter, dev_id, is used primarily for shared interrupt lines. When an interrupt handler is freed (discussed later), dev_id provides a unique cookie to allow the removal of only the desired interrupt handler from the interrupt line. Without this parameter, it would be impossible for the kernel to know which handler to remove on a given interrupt line. You can pass NULL here if the line is not shared, but you must pass a unique cookie if your interrupt line is shared (and unless your device is old and crusty and lives on the ISA bus, there is good chance it must support sharing). This pointer is also passed into the interrupt handler on each invocation. A common practice is to pass the driver's device structure: This pointer is unique and might be useful to have within the handlers and the Device Model.
- On success, request_irq() returns zero. A nonzero value indicates error, in which case the specified interrupt handler was not registered. A common error is -EBUSY, which denotes that the given interrupt line is already in use (and either the current user or you did not specify SA_SHIRQ).
- Note that request_irq() can sleep and therefore cannot be called from interrupt context or other situations where code cannot block.
- On registration, an entry corresponding to the interrupt is created in /proc/irq.
- The function proc_mkdir() is used to create new procfs entries. This function calls proc_create() to set up the new procfs entries, which in turn call kmalloc() to allocate memory
- In a driver, requesting an interrupt line and installing a handler is done via request_irq():

```
if (request_irq(irqn, my_interrupt, SA_SHIRQ, "my_device", dev)) {
        printk(KERN_ERR "my_device: cannot register IRQ %d\n", irqn);
        return -EIO;
}
```

In this example, irqn is the requested interrupt line, my_interrupt is the handler, the line can be shared, the device is named "my_device," and we passed dev for dev_id. On failure, the code prints an error and returns. If the call returns zero, the handler has been successfully installed. From that point forward, the handler is invoked in response to an interrupt. It is important to initialize hardware and register an interrupt handler in the proper order to prevent the interrupt handler from running before the device is fully initialized.

Freeing an Interrupt Handler

- When we unregister our device drivers it is compulsory to free the interrupt handler what we have registered for the device.
- This frees the interrupt line.
- void free_irq(unsigned int irq, void *dev_id)
- If the specified interrupt line is not shared, this function removes the handler and disables the line. If the interrupt line is shared, the handler identified via dev_id is removed, but the interrupt line itself is disabled only when the last handler is removed.
- A call to free_irq() must be made from process context.

Interrupt handling concepts

- Each CPU core has only one interrupt line coming towards it from the Interrupt controller which has n number of interrupt lines.
- When a core is executing an interrupt the interrupt is said to be in active state, let us suppose we get same interrupt immediately, in that case the the new interrupt will be put to pending and the resultant interrupt line will be in active pending state. Only when the active interrupt is cleared this pending one will be entertained for execution.
- Let us suppose we are in such active pending situation and a higher priority interrupt arrives, in this case the lower priority one is temporarily interrupted and higher one is executed, when the higher one is done the lower one is executed. The lower priority one is still active when interrupted as its context remains in the stack.

On a SMP architecture Advanced Programmable Interrupt Controller(**APIC**) is used to route the interrupts from peripherals to the CPU's.

the APIC, based on

1. the routing table,

2. priority of the interrupt,

3. the load on the CPUs(higher busy one is less burdened)

Let us consider a case of same interrupt being received one an SMP system. FOr each core we have an APIC and for external interrupt interface we have one more APIC.

For example, consider a interrupt is received at IRQ line 10, this goes through external APIC,the interrupt is routed to a particular CPU APIC, for now consider CPU0, this interrupt line is masked until the ISR is handled, which means we will not get a interrupt of the same type if ISR execution is in progress, new occurrence will be put to pending state(only 1).

Once ISR is handled, only then the interrupt line is unmasked for future interrupts

How to write an interrupt handler?

- The declaration of interrupt handler:

```
static irqreturn_t intr_handler(int irq, void *dev_id, struct pt_regs *regs)
```

- The first parameter, irq, is the numeric value(10,11,30, e.t.c) of the interrupt line the handler is supposed to service

- The second parameter, dev_id, is a generic pointer to the same dev_id that was given to request_irq() when the interrupt handler was registered. This value should be unique if it is intended for shared interrupt as it can act as a cookie to differentiate between multiple devices using the same interrupt handler.

- The final parameter, regs, holds a pointer to a structure containing the processor registers and state before servicing the interrupt.

- The return value of an interrupt handler is the special type irqreturn_t. An interrupt could be serviced or could not be An interrupt handler can return two special values, IRQ_NONE or IRQ_HANDLED. The former is returned when the interrupt handler detects an interrupt for which its device was not the originator. The latter is returned if the interrupt handler was correctly invoked, and its device did indeed cause the interrupt. Alternatively, IRQ_RETVAL(val) may be used. If val is non-zero, this macro returns IRQ_HANDLED. Otherwise, the macro returns IRQ_NONE. These special values are used to let the kernel know whether devices are issuing spurious (that is, unrequested) interrupts. If all the interrupt handlers on a given interrupt line return IRQ_NONE, then the kernel can detect the problem. Note the curious return type, irqreturn_t, which is simply an int

- The interrupt handler is normally marked static because it is never called directly from another file.

- The role of the interrupt handler depends entirely on the device and its reasons for issuing the interrupt. At a minimum, most interrupt handlers need to provide acknowledgment to the device that they received the interrupt. Devices that are more complex need to additionally send and receive data and perform extended work in the interrupt handler. As mentioned, the extended work is pushed as much as possible into the bottom half handler, which I have discussed in other post on Bottom Halves.

Shared Handlers

A shared handler is registered and executed much like a non-shared handler. There are three main differences:

- The SA_SHIRQ flag must be set in the flags argument to request_irq().
- The dev_id argument must be unique to each registered handler. A pointer to any per-device structure is sufficient; a common choice is the device structure as it is both unique and potentially useful to the handler. You cannot pass NULL for a shared handler!
- The interrupt handler must be capable of distinguishing whether its device actually generated an interrupt. This requires both hardware support and associated logic in the interrupt handler. If the hardware did not offer this capability, there would be no

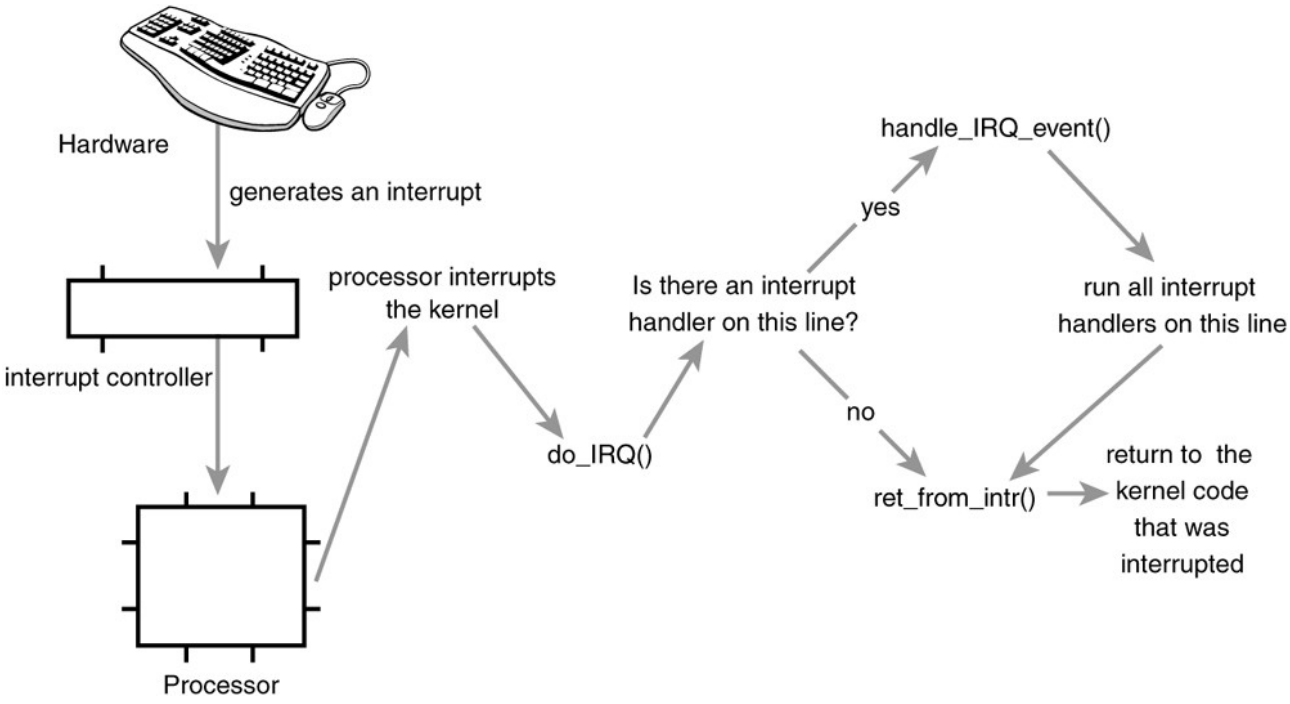
way for the interrupt handler to know whether its associated device or some other device sharing the line caused the interrupt.

Interrupt Context

- When executing an interrupt handler or bottom half, the kernel is in interrupt context. (process context is the mode of operation the kernel is in while it is executing on behalf of a process for example, executing a system call or running a kernel thread.)
- In process context, the current macro points to the associated task. Furthermore, because a process is coupled to the kernel in process context, process context can sleep or otherwise invoke the scheduler.
- Interrupt context is not associated with a process. The current macro is not relevant (although it points to the interrupted process). Without a backing process, interrupt context cannot sleephow would it ever reschedule? Therefore, you cannot call certain functions from interrupt context. If a function sleeps, you cannot use it from your interrupt handlerthis limits the functions that one can call from an interrupt handler.
- Interrupt context is time critical because the interrupt handler interrupts other code. Code should be quick and simple. Busy looping is discouraged. This is a very important point; always keep in mind that your interrupt handler has interrupted other code (possibly even another interrupt handler on a different line!).
- Because of this asynchronous nature, it is imperative that all interrupt handlers be as quick and as simple as possible. As much as possible, work should be pushed out from the interrupt handler and performed in a bottom half, which runs at a more convenient time.
- The setup of an interrupt handler's stacks is a configuration option. Historically, interrupt handlers did not receive their own stacks. Instead, they would share the stack of the process that they interrupted.
- The kernel stack is two pages in size; typically, that is 8KB on 32-bit architectures and 16KB on 64-bit architectures. Because in this setup interrupt handlers share the stack, they must be exceptionally economical with what data they allocate there. Of course, the kernel stack is limited to begin with, so all kernel code should be cautious.

Implementation of Interrupt Handling

The implementation of interrupt handler is architecture dependent and hardware dependent.



A device issues an interrupt by sending an electric signal over its bus to the interrupt controller.

- If the interrupt line is enabled (they can be masked out), the interrupt controller sends the interrupt to the processor.
- In most architectures, this is accomplished by an electrical signal that is sent over a special pin to the processor. Unless interrupts are disabled in the processor (which can also happen), the processor immediately stops what it is doing, disables the interrupt system, and jumps to a predefined location in memory and executes the code located there. This predefined point is set up by the kernel and is the entry point for interrupt handlers.
- The interrupt's journey in the kernel begins at this predefined entry point, just as system calls enter the kernel through a predefined exception handler.
- For each interrupt line, the processor jumps to a unique location in memory and executes the code located there.
- In this manner, the kernel knows the IRQ number of the incoming interrupt. The initial entry point simply saves this value and stores the current register values (which belong to the interrupted task) on the stack; then the kernel calls do_IRQ(). From here onward, most of the interrupt handling code is written in C however, it is still architecture dependent.
- The do_IRQ() function is declared as

unsigned int do_IRQ(struct pt_regs regs)

- Because the C calling convention places function arguments at the top of the stack, the pt_regs structure contains the initial register values that were previously saved in the assembly entry routine. Because the interrupt value was also saved, do_IRQ() can extract it.
- The x86 code is int irq = regs.orig_eax & 0xff;
- After the interrupt line is calculated, do_IRQ() acknowledges the receipt of the interrupt and disables interrupt delivery on the line. On normal PC machines, these operations are handled by mask_and_ack_8259A(), which do_IRQ() calls.
- Next, do_IRQ() ensures that a valid handler is registered on the line, and that it is enabled and not currently executing. If so, it calls handle_IRQ_event() to run the installed interrupt handlers for the line. On x86, handle_IRQ_event() is

```
asmlinkage int handle_IRQ_event(unsigned int irq, struct pt_regs *regs,
                        struct irqaction *action)
{
        int status = 1;
        int retval = 0;

        if (!(action->flags & SA_INTERRUPT))
                local_irq_enable();

        do {
                status |= action->flags;
                retval |= action->handler(irq, action->dev_id, regs);
                action = action->next;
        } while (action);

        if (status & SA_SAMPLE_RANDOM)
                add_interrupt_randomness(irq);

        local_irq_disable();

        return retval;
}
```

- First, because the processor disabled interrupts, they are turned back on unless SA_INTERRUPT was specified during the handler's registration. Recall that

SA_INTERRUPT specifies that the handler must be run with interrupts disabled. Next, each potential handler is executed in a loop.

- If this line is not shared, the loop terminates after the first iteration. Otherwise, all handlers are executed. After that, add_interrupt_randomness() is called if SA_SAMPLE_RANDOM was specified during registration.
- This function uses the timing of the interrupt to generate entropy for the random number generator.
- Finally, interrupts are again disabled (do_IRQ() expects them still to be off) and the function returns. Back in do_IRQ(), the function cleans up and returns to the initial entry point, which then jumps to ret_from_intr().
- The routine ret_from_intr() is, as with the initial entry code, written in assembly. This routine checks whether a reschedule is pending (this implies that need_resched is set).
- If a reschedule is pending, and the kernel is returning to user-space (that is, the interrupt interrupted a user process), schedule() is called. If the kernel is returning to kernel-space (that is, the interrupt interrupted the kernel itself), schedule() is called only if the preempt_count is zero (otherwise it is not safe to preempt the kernel). After schedule() returns, or if there is no work pending, the initial registers are restored and the kernel resumes whatever was interrupted
- On x86, the initial assembly routines are located in arch/i386/kernel/entry.S and the C methods are located in arch/i386/kernel/irq.c. Other supported architectures are similar.

/proc/interrupts

- Procfs is a virtual filesystem that exists only in kernel memory and is typically mounted at /proc.
- Reading or writing files in procfs invokes kernel functions that simulate reading or writing from a real file.
- A relevant example is the /proc/interrupts file, which is populated with statistics related to interrupts on the system. Here is sample output from a uniprocessor PC:

```
    CPU0
 0:   3602371   XT-PIC   timer
 1:   3048      XT-PIC   i8042
 2:   0         XT-PIC   cascade
 4:   2689466   XT-PIC   uhci-hcd, eth0
 5:   0         XT-PIC   EMU10K1
12:  85077     XT-PIC   uhci-hcd
15:  24571     XT-PIC   aic7xxx
```

NMI: 0
LOC: 3602236
ERR: 0

- The first column is the interrupt line. On this system, interrupts numbered 02, 4, 5, 12, and 15 are present
- . Handlers are not installed on lines not displayed.
- The second column is a counter of the number of interrupts received. A column is present for each processor on the system, but this machine has only one processor.
- As you can see, the timer interrupt has received 3,602,371 interrupt, whereas the sound card (EMU10K1) has received none (which is an indication that it has not been used since the machine booted).
- The third column is the interrupt controller handling this interrupt. XT-PIC corresponds to the standard PC programmable interrupt controller. On systems with an I/O APIC, most interrupts would list IO-APIC-level or IO-APIC-edge as their interrupt controller.
- Finally, the last column is the device associated with this interrupt. This name is supplied by the devname parameter to request_irq(), as discussed previously. If the interrupt is shared, as is the case with interrupt number four in this example, all the devices registered on the interrupt line are listed.
- procfs code is located primarily in fs/proc. The function that provides /proc/interrupts is, not surprisingly, architecture dependent and named show_interrupts().

System Calls

In this post we will discuss mainly about what are system calls, why do we need it and how to implement it.

What is a system call ?

To understand this first we would ask ourselves what are the stuffs the OS(read kernel) needs to do ?

- Process Management (starting, running, stopping processes)
- File Management(creating, opening, closing, reading, writing, renaming files)
- Memory Management (allocating, deallocating memory)
- Other stuff (timing, scheduling, network management).

So, system call is an interface through which user space applications request the Kernel do perform the operations listed above.

An example would be , the user space requests to open a device(hardware).

In short we can say that the System call is an interface between user space processes and hardware.

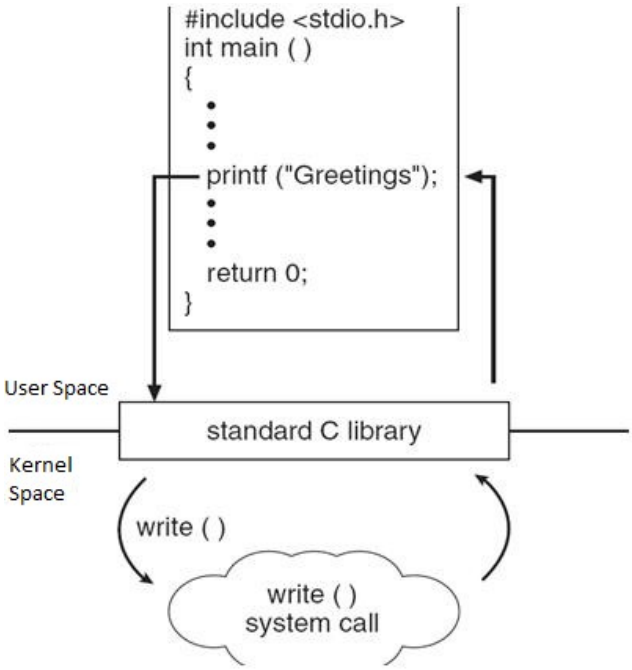
Why do we need system call?

1. It provides an abstraction to the user space process. Eg. open call for user means just open the device, the user doesn't need to care about intricacy of the call.
2. It maintains the system security and stability as the kernel first checks the authenticity of the call before requesting it a service.
3. It helps in virtualization of various processes i.e various processes can use it independently.

System call interface and C library.

The system call interface in Linux, as with most Unix systems, is provided in part by the C library.

We will see How System call works using a example of printf() call in userspace.



Syscalls

- System calls (*syscalls* in Linux) are accessed via function calls. System calls need inputs and also provide a return value (*long)* signifies success or error.(0 generally means success).
- System calls have a defined behavior.

For example, the system call getpid() is defined to return an integer that is the current process's PID.

The implementation of this syscall in the kernel is very simple:

```
asmlinkage long sys_getpid(void)

{
        return current->tgid;

}
```

Some important observations from this-

A convention in which a system call is appended with sys in kernel space.
asmlinkage modifier -tells the compiler that the function should not expect to find any of its arguments in registers (a common optimization), but only on the CPU's stack.

- In Linux, each system call is assigned a *syscall number*. This is a unique number that is used to reference a specific system call.
- When the syscall number is assigned, it cannot changed or be recycled.
- System calls in Linux are faster than in many other operating systems. (such as fast context switch times.
- The kernel keeps track of all the registered system calls in table sys_call_table which is defined in enTRy.S(assembler file) in arch/arch-name/kernel/

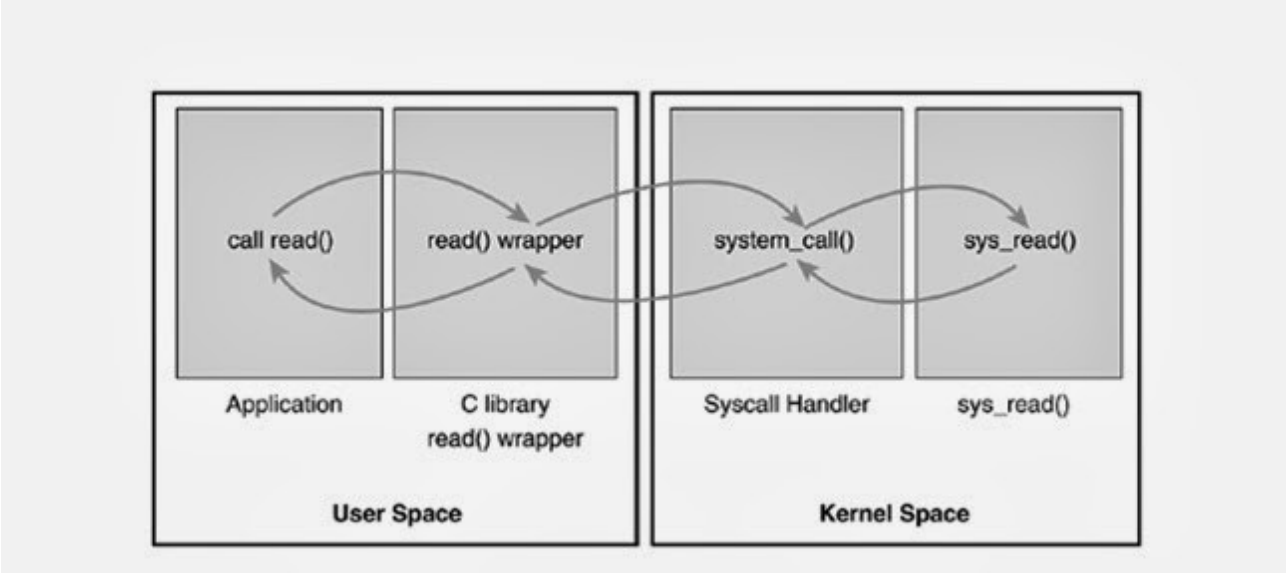
System Call Handler:-

- Since the system call code lies in kernel side, so to execute it we must switch the processor to kernel mode when system call is executed.
- This is done by issuing a software interrupt.
- In this mechanism an exception is raised and the Kernel switches to kernel mode and execute the system call handler.
- The defined software interrupt on x86 is the **int $0x80** instruction in ARM the address is 0x08 offset from start of exception vector base(0X00000000, or 0xFFFF0000)
- It triggers a switch to kernel mode and the execution of exception vector 128, which is the system call handler.
- The system call handler function is system_call().
- It is architecture dependent and typically implemented in assembly in **entry.S**
- User space first enters the system call number in eax register(X86) and causes the trap.

- The kernel reads the value of the eax register and calls the appropriate system call handler.
- The system_call() function checks the validity of the given system call number by comparing it to NR_syscalls.
- If it is larger than or equal to NR_syscalls, the function returns -ENOSYS. Otherwise, the specified system call is invoked:

```
call *sys_call_table(,%eax,4)
```

- Because each element in the system call table is 32 bits (four bytes), the kernel multiplies the given system call number by four to arrive at its location in the system call table



Now, the system call is called with some parameters, generally upto 5 parameters, we store the parameters values in registers **ebx, ecx, edx, esi**, and edi.

- In some unique cases when 6 or more parameters are passed then a single register is used which stores the pointer to the user space where all the parameters are stored.
- Not only this, even the return value is stored in the the register(eax in case of X86).

How to implement system calls?

Adding a system call is an easy task. But it is the implementation that has to be done carefully.

Now we will see what are the steps used to implement a system call.

First we must define its purpose. What is the use of this system call? The syscall should have exactly one purpose.

Next, we must define system call's arguments, return value, and error codes.

The system call should have a clean and simple interface with the smallest number of arguments possible.

- Final Steps in Binding a System Call

1. First, add an entry to the end of the system call table.
2. For each architecture supported, the syscall number needs to be defined in *<asm/unistd.h>*.
3. The syscall needs to be compiled into the kernel image

How system call verifies parameters(arguments)?

- System calls must make sure all of their parameters are valid and legal. Such as *access permission*.
- System calls must carefully verify all their parameters to ensure that they are valid and legal.
- The system call runs in kernel-space, and if the user is able to pass invalid input into the kernel without restraint, the system's security and stability can suffer, in short the kernel can be hacked!!
- For example, for file I/O syscalls, the syscall must check whether the file descriptor is valid. Process-related functions must check whether the provided PID is valid. Every parameter must be checked to ensure it is not just valid and legal, but correct.
- One of the most important checks is the validity of any pointers that the user provides. Imagine if a process could pass any pointer into the kernel, unchecked, with warts and all, even passing a pointer for which it did not have read access! Processes could then trick the kernel into copying data for which they did not have access permission, such as data belonging to another process. Before following a pointer into user-space, the system must ensure that

1. The pointer points to a region of memory in user-space. Processes must not be able to trick the kernel into reading data in kernel-space on their behalf.

2. The pointer points to a region of memory in the process's address space. The process must not be able to trick the kernel into reading someone else's data.
3. If reading, the memory is marked readable. If writing, the memory is marked writable. The process must not be able to bypass memory access restrictions

- Two methods for performing the requisite checks and the desired copy to and from user-space:
  1. For writing into user-space, the method **copy_to_user***(destination memory address , source pointer , size of the data to copy)* is provided.
  2. For reading from user-space, the method **copy_from_user***(destination memory address , source pointer, the number from the second parameter reading into the first parameter)* is used.
  - Both of these functions return the number of bytes they failed to copy on error. On success, they return zero. It is standard for the syscall to return -EFAULT in the case of such an error.
  - check is for valid permission. A call to capable() with a valid capabilities flag returns nonzero if the caller holds the specified capability and zero otherwise. For example, capable(CAP_SYS_NICE) checks whether the caller has the ability to modify nice values of other processes.

Process Management

What are processes and threads??

- A process is simply an abstraction of a running program.
- Processes include a set of resources such as open files and pending signals, internal kernel data, processor state, an address space, one or more threads of execution, and a data section containing global variables.

- Threads are the units of execution within a program.

- In Linux Operating system, the scheduling of thread or processes take place with a common concept of tasks--

A Linux Scheduler will schedule tasks and these tasks could be

1. a single-threaded process (e.g. created by `fork` without any thread library)
2. any thread inside a multi-threaded process (including its main thread)
3.kernel tasks, which are started internally in the kernel and stay in kernel space (e.g. `kworke,kswapd` etc...)

- Each thread within a process has a 1. unique program counter, 2. process stack, 3. and set of processor registers.

- Threads are light weight. They don't have their own memory spaces and other resources unlike processes. All processes start with a single thread. So they behave like lightweight processes but are always tied to a parent "thick" process. So, creating a new process is a slightly heavy task and involves allocating all these resources while cresting a thread does not. Killing a process also involves releasing all these resources while a thread does not. However, killing a thread's parent process releases all resources of the thread.

- A process is suspended by itself and resumed by itself. Same with a thread but if a thread's parent process is suspended then the threads are all suspended.

- In Linux, a process is created by means of the fork() system call, which creates a new process by duplicating an existing one.
- The new process is an exact copy of the old process. (Now, a question might come immediately to our mind- who creates the first process?? The first process is the init process which is literally created from scratch during booting).
- The process that calls fork() is the parent, whereas the new process is the child. The parent resumes execution and the child starts execution at the same place, where the call returns. The fork() system call returns from the kernel twice: once in the parent process and again in the newborn child(see the below diagram).
-

fork()

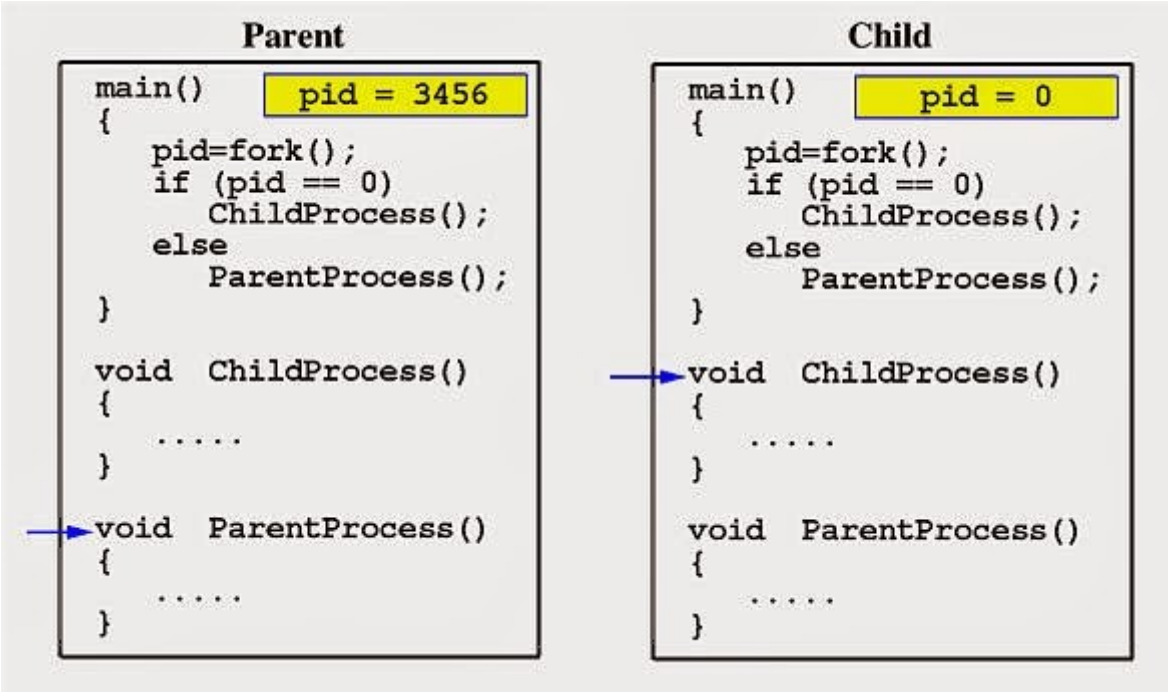
- The fork() function is used to create a new process by duplicating the existing process from which it is called.
- The existing process from which this function is called becomes the parent process and the newly created process becomes the child process.

- Inside the newly created address space exec*() family of function calls is used to create a new address space and load a new program into it. In modern Linux kernels, fork() is actually implemented via the clone() system call,
- Finally, a program exits via the exit() system call. This function terminates the process and frees all its resources.
- Fork is the only mechanism to create new process, any other mechanism will inherently use fork inside.
- A parent process can inquire about the status of a terminated child via the wait4()system call, which enables a process to wait for the termination of a specific process.
- The child is a duplicate copy of the parent but there are some exceptions to it.

1. ***The child has a unique PID like any other process running in the operating system.***
2. ***The child has a parent process ID which is same as the PID of the process that created it.***
3. ***Resource utilization and CPU time counters are reset to zero in child process.***
4. ***Set of pending signals in child is empty.***
5. ***Child does not inherit any timers from its parent***
6. ***The child does not inherit its parent's memory locks (mlock(2), mlockall(2))***
7. ***The child does not inherit outstanding asynchronous I/O operations from its parent.***

What is the return value of fork()?

- If the fork() function is successful then it returns twice.
- Once it returns in the child process with return value zero and then it returns in the parent process with child's PID as return value.
- This behavior is because of the fact that once the fork is called, child process is created and since the child process shares the text segment with parent process and continues execution from the next statement in the same text segment so fork returns twice (once in parent and once in child).



fork() vs vfork() vs exec() vs system() vs clone()

Let us first see the standard definition of these system calls.

Fork : The fork call is used to duplicate the current process, the new process identical in almost every way except that it has its own PID. The return value of the function fork distinguishes the two processes, zero is returned in the child and PID of child in parent process.

Exec : The exec call is a way to basically replace the entire current process with a new program. It loads the program into the current process space and runs it from the entry point. As a new process is not created, the process identifier (PID) does not change, but the machine code, data, heap, and stack of the process are replaced by those of the new program.exec() replaces the current process with a the executable pointed by the function. Control never returns to the original program unless there is an exec() error. exec system call can be executed as `execl, execlp, execle, execv, execvp, execvpe`

Vfork : The basic difference between vfork and fork is that when a new process is created with vfork(), the parent process is temporarily suspended, and the child process might borrow the parent's address space. This strange state of affairs continues until the child process either exits, or calls execve(), at which point the parent process continues. This means that the child process of a vfork() must be careful to avoid unexpectedly modifying variables of the parent process. In particular, the child process must not return from the function containing the vfork() call, and it must not call exit() (if it needs to exit, it should use _exit(); actually, this is also true for the child of a normal fork()).

The intent of `vfork` was to eliminate the overhead of copying the whole process image if you only want to do an `exec*` in the child. Because `exec*` replaces the whole image of the child process, there is no point in copying the image of the parent.

```
if ((pid = vfork()) == 0) {
```

```
    execl(..., NULL); /* after a successful execl the parent should be resumed */
    _exit(127); /* terminate the child in case execl fails */
}
```

For other kinds of uses, `vfork` is dangerous and unpredictable.
With most current kernels, however, including Linux, the primary benefit of `vfork` has disappeared because of the way `fork` is implemented. Rather than copying the whole image when `fork` is executed, copy-on-write techniques are used.

Clone : Clone, as fork, creates a new process. Unlike fork, these calls allow the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers.
When the child process is created with clone, it executes the function application fn(arg). (This differs from fork, where execution continues in the child from the point of the original fork call.) The fn argument is a pointer to a function that is called by the child process at the beginning of its execution. The arg argument is passed to the fn function.
When the fn(arg) function application returns, the child process terminates. The integer returned by fn is the exit code for the child process. The child process may also terminate explicitly by calling exit(2) or after receiving a fatal signal.

System : The **system**() library function uses fork(2) to create a child process that executes the shell command specified in *command* using execl(3) as follows:

```
execl("/bin/sh", "sh", "-c", command, (char *) 0);
```

system() returns after the command has been completed.During execution of the command

, **SIGCHLD** will be blocked, and **SIGINT** and **SIGQUIT** will be ignored, in the process that calls **system**()

(these signals will be handled according to their defaults inside the child process that executes *command*).
If *command* is NULL, then **system**() returns a status indicating whether a shell is available on the system

- In situations where performance is critical and/or memory limited, `vfork` + `exec*` can therefore be a good alternative to `fork` + `exec*`. The problem is that it is less safe and the man page says `vfork` is likely to become deprecated in the future.
- Because memory page tables are not duplicated, `vfork` is much faster than `fork` and `vfork`'s execution time is not affected by the amount of memory the parent process uses
- system() will invoke your systems default command shell, which will execute the command string passed as an argument, that itself may or may not create further processes, that would depend on the command and the system. Either way, at least a command shell process will be created.
- With system() you can invoke any command, whereas with exec(), you can only invoke an executable file. Shell scripts and batch files must be executed by the command shell.

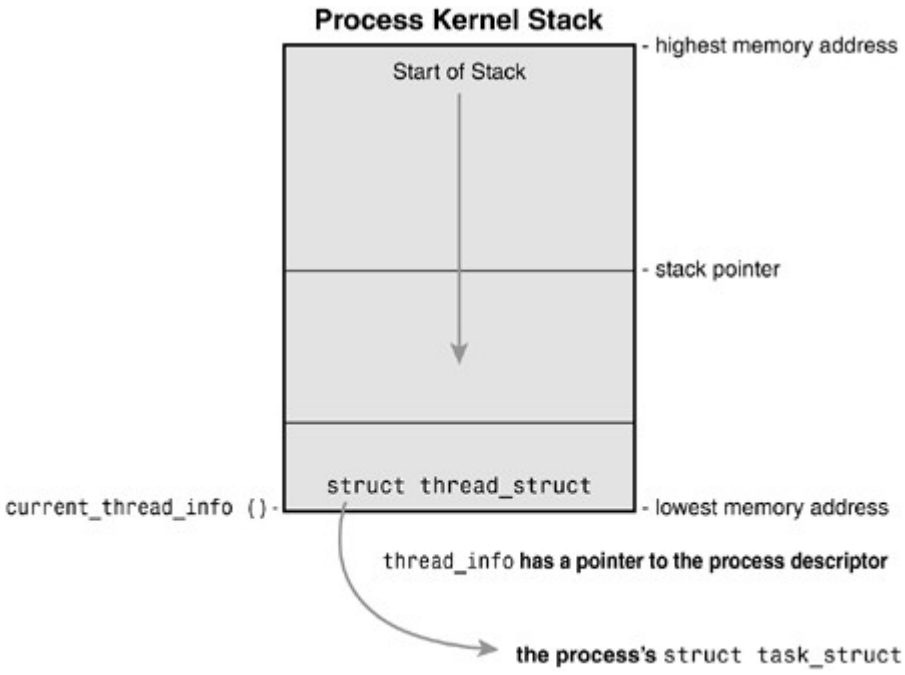
Process Descriptor and the Task Structure

- The kernel stores the list of processes in a circular doubly linked list called the task list.
- **Process descriptor is nothing but each element of this task list** of the type **struct task_struct**, which is defined in <linux/sched.h>. The process descriptor contains all the information about a specific process.
- Some texts on operating system design call this list the task array. Because the Linux implementation is a linked list and not a static array, it is called the task list.
- The task_struct is a relatively large data structure, at around 1.7 kilobytes on a 32-bit machine.
- This size, however, is quite small considering that the structure contains all the information that the kernel has and needs about a process.
- The process descriptor contains the data that describes the executing program open files, the process's address space, pending signals, the process's state, and much more.
- This linked list is stored in kernel space.
- There is one more structure, thread_info which holds more architecture-specific data than the task_struct.

Allocating the Process Descriptor

- Threads in Linux are treated as processes that just happen to share some resources. Each thread has its own thread_info. There are two basic reasons why there are two such structures
  1. thread_info is architecture dependent. task_struct is generic.
  2. thread_info consumes the space of the kernel stack for that process, so it should be kept small.

thread_info is placed at the bottom of the stack as a micro-optimization that makes it possible to compute its address from the current stack pointer by rounding down by the stack size saving a CPU register.


Process Kernel Stack
- highest memory address
Start of Stack
- stack pointer
struct thread_struct
current_thread_info ()
- lowest memory address
thread_info has a pointer to the process descriptor
the process's struct task_struct