

System Call Design and Implementation

(Please go through links in this passage)

1 . System Call Concept

https://en.wikipedia.org/wiki/System_call

http://www.linfo.org/system_call.html

2 . System Call functioning procedures

<http://www.cs.columbia.edu/~jae/4118-LAST/L02-intro2-osc-ch2.pdf>

https://en.wikipedia.org/wiki/Linux_kernel_interfaces

2.1 System call in Linux

<http://man7.org/linux/man-pages/man2/syscalls.2.html>

http://faculty.salina.k-state.edu/tim/oss/Introduction/sys_calls.html

Linux 2.6.35.4 system calls:

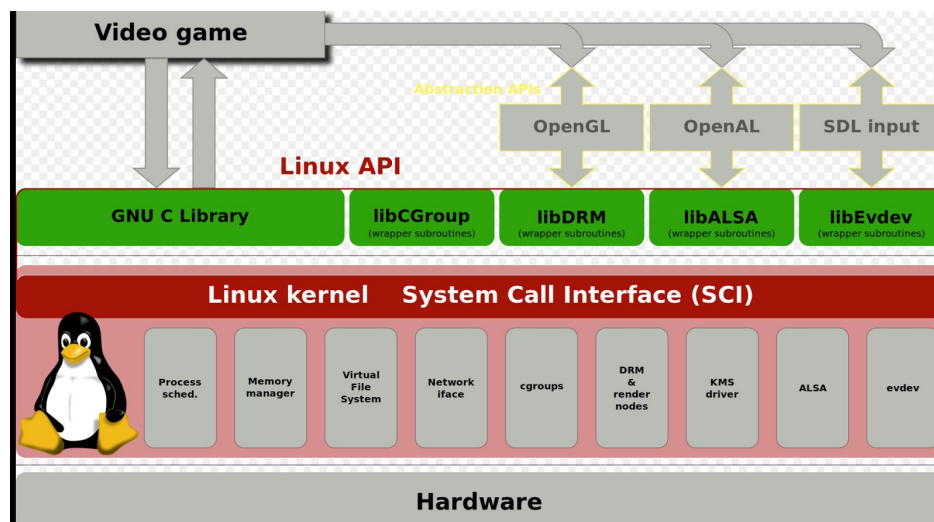
<http://syscalls.kernelgrok.com/>

GNU C Library:

https://en.wikipedia.org/wiki/GNU_C_Library

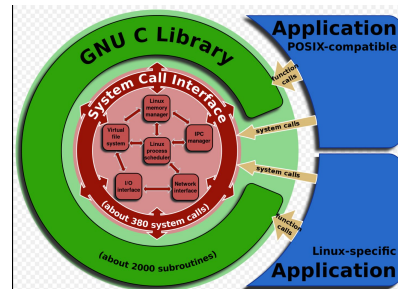
Linux kernel interfaces:

https://en.wikipedia.org/wiki/Linux_kernel_interfaces#Linux_API



Linux API [\[edit \]](#)

The Linux API is the kernel–user space API, which allows programs in user space to access system resources and services of the Linux kernel.^[3] It is composed out of the System Call Interface of the Linux kernel and the subroutines in the [GNU C Library](#) (glibc). The focus of the development of the Linux API has been to provide the *usable features* of the specifications defined in [POSIX](#) in a way which is reasonably compatible, robust and performant, and to provide additional useful features not defined in POSIX, just as the kernel–user space APIs of other systems implementing the POSIX API also provide additional features not defined in POSIX.



In many cases, the C library wrapper function does nothing more than:

- * copying arguments and the unique system call number to the registers where the kernel expects them;
- * trapping to kernel mode, at which point the kernel does the real work of the system call;
- * setting `errno` if the system call returns an error number when the kernel returns the CPU to user mode.

RETURN VALUE [top](#)

On error, most system calls return a negative error number (i.e., the negated value of one of the constants described in [errno\(3\)](#)). The C library wrapper hides this detail from the caller: when a system call returns a negative value, the wrapper copies the absolute value into the `errno` variable, and returns -1 as the return value of the wrapper.

The value returned by a successful system call depends on the call. Many system calls return 0 on success, but some can return nonzero values from a successful call. The details are described in the individual manual pages.

In some cases, the programmer must define a feature test macro in order to obtain the declaration of a system call from the header file specified in the man page SYNOPSIS section. (Where required, these feature test macros must be defined before including *any* header files.) In such cases, the required macro is described in the man page. For further information on feature test macros, see [feature_test_macros\(7\)](#).

Linux Syscall Reference										
Show 100 entries		Search:								
#	Name	eax	ebx	ecx	edx	esi	edi	Registers		
0	sys_restart_syscall	0x00	-	-	-	-	-	-	-	kernel/signal.c:2058
1	sys_exit	0x01	int error_code	-	-	-	-	-	-	kernel/exit.c:1046
2	sys_fork	0x02	struct pt_regs *	-	-	-	-	-	-	arch/alpha/kernel/entry.5:716
3	sys_read	0x03	unsigned int fd	char __user *buf	size_t count	-	-	-	-	fs/read_write.c:391
4	sys_write	0x04	unsigned int fd	const char __user *buf	size_t count	-	-	-	-	fs/read_write.c:408
5	sys_open	0x05	const char __user *filename	int flags	int mode	-	-	-	-	fs/open.c:900
6	sys_close	0x06	unsigned int fd	-	-	-	-	-	-	fs/open.c:969
7	sys_waitpid	0x07	pid_t pid	int __user *stat_addr	int options	-	-	-	-	kernel/exit.c:1771
8	sys_creat	0x08	const char __user *pathname	int mode	-	-	-	-	-	fs/open.c:933
9	sys_link	0x09	const char __user *oldname	const char __user *newname	-	-	-	-	-	fs/namei.c:2520
10	sys_unlink	0x0a	const char __user *pathname	-	-	-	-	-	-	fs/namei.c:2352

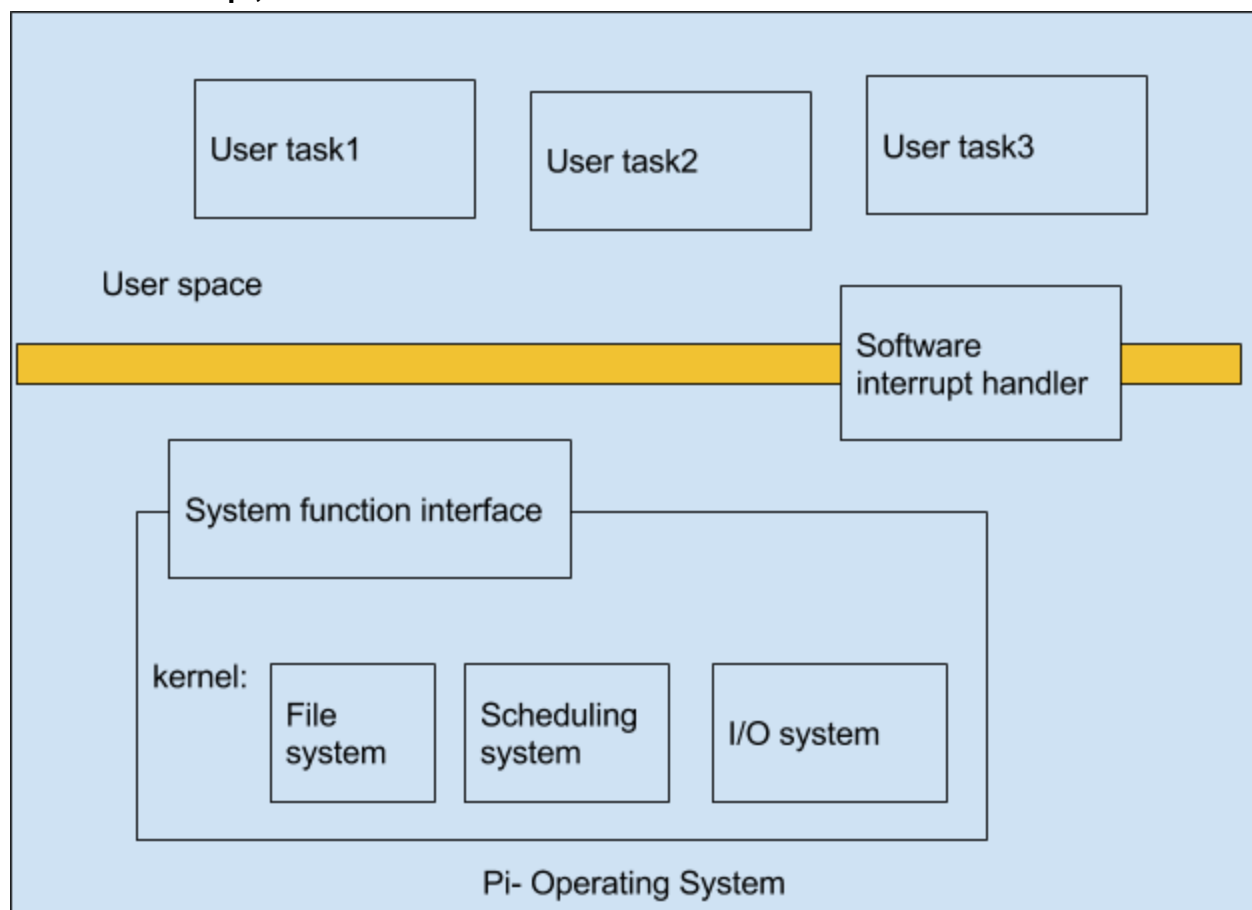
From: <http://syscalls.kernelgrok.com/>

2.2. System call in pi-OS

0-337 compatible with linux system call;

1000- 1100 pi-OS specific system call

The figure below shows main components in pi-OS, user task task privileged action via software interrupt, and handler will further call some kernel functions:



Mode bits:

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0290g/I27695.html>

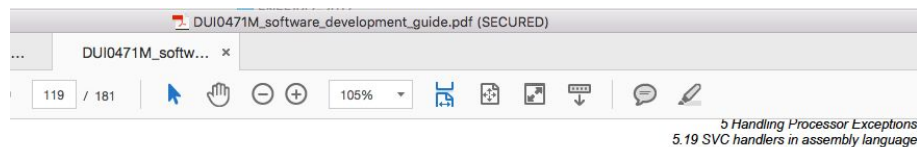
<http://stackoverflow.com/questions/22295566/how-can-i-put-arm-processor-in-different-modes-using-c-program>

Return from exception:

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0040d/Cacbacic.html>

SVC handler:

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0203j/Cacdfeci.html>



5.19 SVC handlers in assembly language

The easiest way to call the handler for the requested SVC number is to use a jump table.

SVC jump table

```
AREA SVC_Area, CODE, READONLY
PRESERVE8
IMPORT SVCOutOfRange
IMPORT MaxSVC
CMP R0, #MaxSVC ; Range check
LDRLS pc, [pc, R0, LSL #2]
B SVCOutOfRange
SVCJumpTable
DCD SVCnum0
DCD SVCnum1
; DCD for each of other SVC routines
; SVC number 0 code
SVCnum0 B EndofSVC
; SVC number 1 code
SVCnum1 B EndofSVC
; Rest of SVC handling code
EndofSVC
; Return execution to top level
; SVC handler so as to restore
; registers and return to program.
END
```

If R0 contains the SVC number, the code in the preceding example can be inserted into the following example, after the BIC instruction.

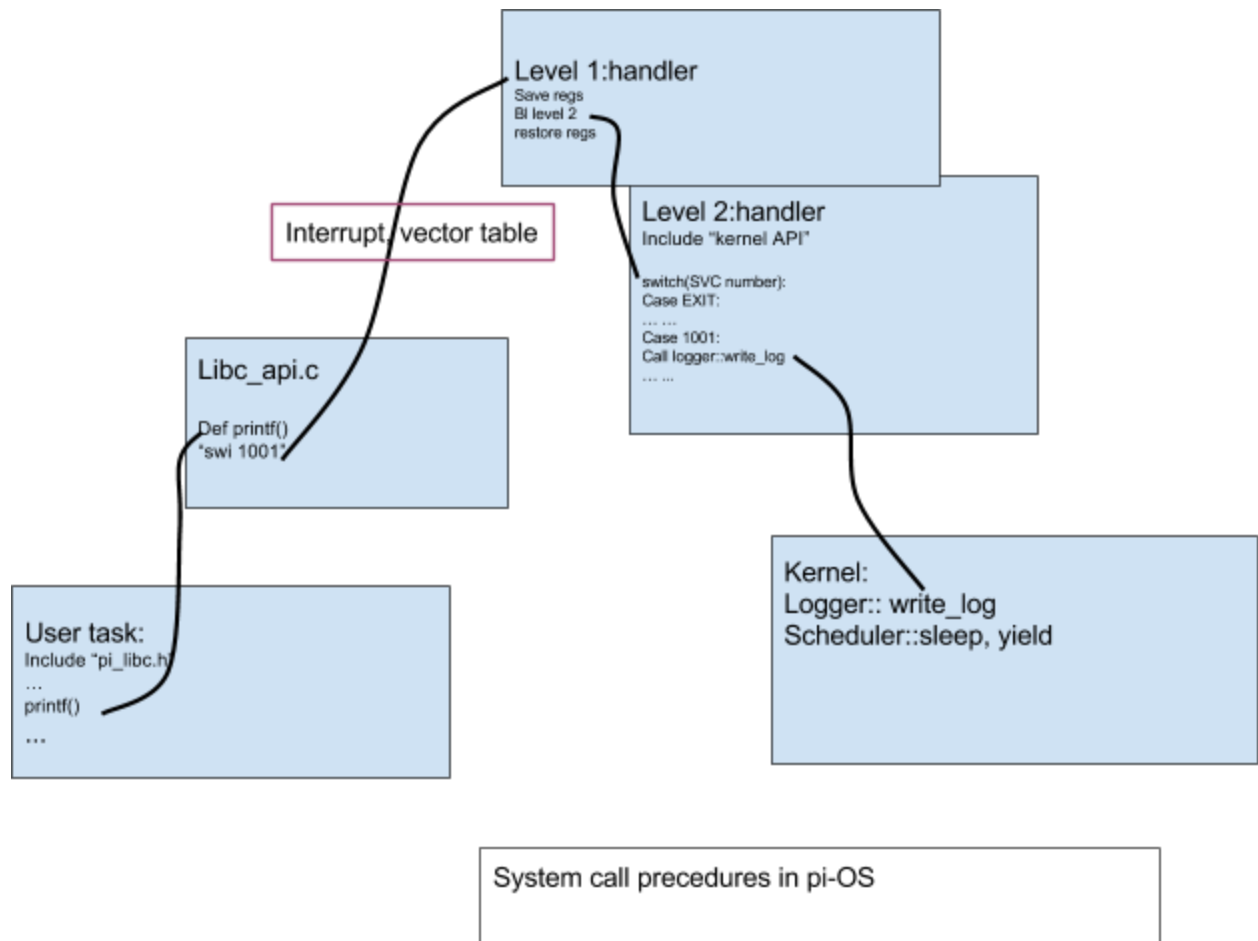
Top-level SVC handler

```
PRESERVE8
AREA TopLevelSVC, CODE, READONLY ; Name this block of code.
EXPORT SVC_Handler
SVC_Handler
PUSH {R0-R12, lr} ; Store registers.
LDR R0, [lr, #-4] ; Calculate address of SVC
; instruction and load it
; into R0.
BIC R0, R0, #0xFF000000 ; Mask off top 8 bits of
; instruction to give SVC number.
; Use value in R0 to determine which SVC routine to execute.
; LDM sp!, {R0-R12, pc}^ ; Restore registers and return.
END
```

Please refer software development guide.pdf for exception handling details.

2.3 system call working procedures in pi-OS

When the user task want to use kernel functions(such as printf or sleep), it need to call libc function. Libc function may preprocess this call a little, then it will make a system call. This trigger system to be in SUPERVISOR mode. CPU then goes to vector table and then branch to first level handler. First level handler would save USER mode registers, find out SVC number, branch to second level handler, after second level handler return, it would restore CPU registers and switch to user mode.



2.4 System call table for pi-OS

Below is a system call table in pi-OS. Note that we put three parameters in r0, r1, r2. And we put system call number in r3. This is our design in pi-OS, just to be compatible with GCC compiling convention (So we can take this advantage to pass r0, r1, r2, r3 directly to level 2 handler's C function). By GCC compiling convention, return value of a C function is stored in r0. So we can pass return value using r0. If you need a return value for a system call, you just read r0 after SVC/SWI instruction.

System call Func	Arg1	Arg2	Arg3	Arg4
ARM-none-eabi	r0	r1	r2	r3(SVC number)
restart				0
exit	Int errNUM			1
fork	Int pid			2

read	File * fp	Char * buffer	int num	3
write	File * fp	Char * buffer	int num	4
open	Char * path	Int mode		5
close	File * fp			6
printf	Char * message			1001
sleep	Int time			1002
getKernelRegPtr				1003
getCurrentTaskPtr				1004

2.5 parameters passing and returning value

Example: a system call `pi_lib_test`. Suppose that we have a C library function `pi_lib_test`, which take three parameters and has one return value. In `task1_run`, it will call `pi_lib_test(2, 3,4)`. 2nd level system call handler for this function do some mathematical computation, then return the result.

In `task1_run`, compiler would put 2,3,4 into `r0`, `r1`, `r2`. When system call handler executes, it will get parameters from these three registers. It also put return value in `r0`. When CPU switch back to user mode, `task1_run` can just get return value from `r0`.

```

145 //system call number: 1000;
146 int pi_lib_test(int a, int b, int c){
147     int res = 0;
148     asm("swi 1000");
149     __asm ("mov %[output], r0"
150           : [output] "=r" (res)
151           );
152     return res;
153 }
154 |

```

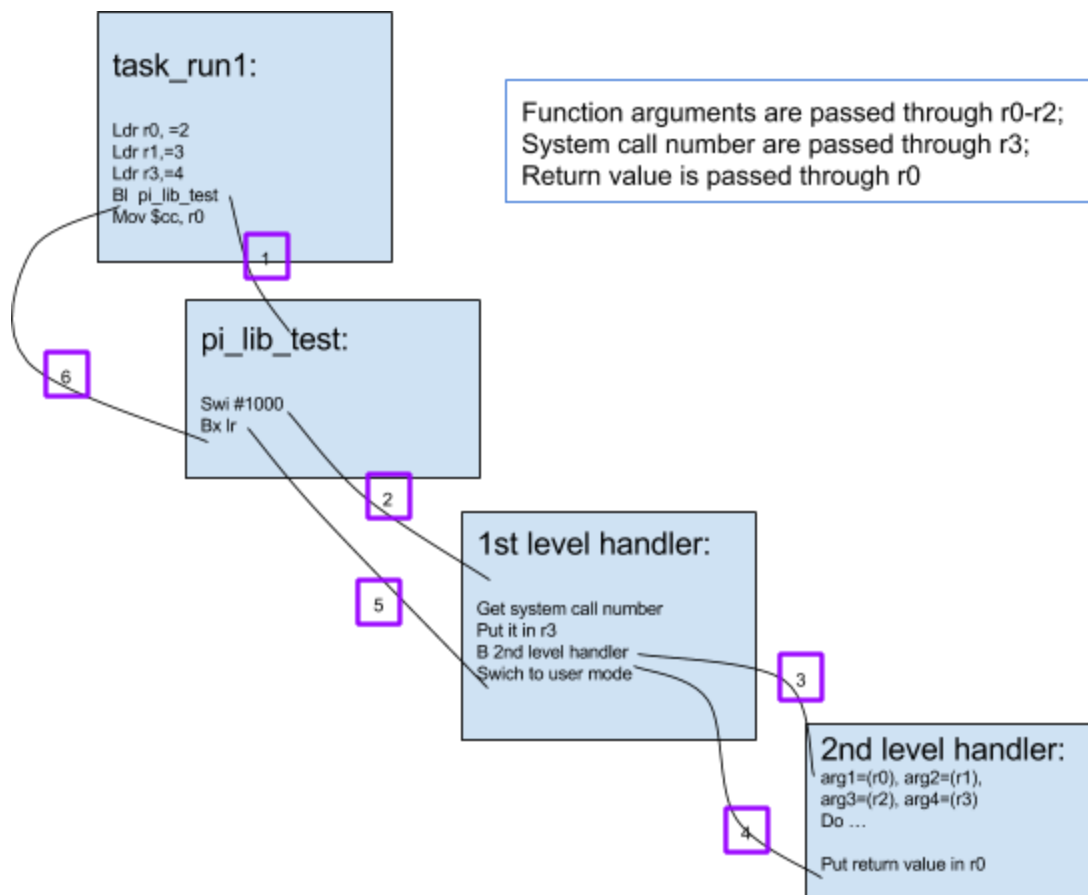


```

59 void task1_run()
60 {
61     printf("Hello From task1 . ----");
62     int cc = pi_lib_test(2, 3, 4);
63     printf("Result of pi_lib_test is %d", cc);
64
65     printf("Task1 has been done.----");
66 }
67

19 int SWIHandler (unsigned r0, unsigned r1, unsigned r2, unsigned r3)
20 {
21     // ...
22     pKernel->write_log("r0=%d,r1=%d,r2=%d,r3=%d", r0, r1, r2, r3);
23
24     switch(r3){
25     case 1:
26         // ...
27         // ...
28         break;
29     case 1000: //printf(char *)
30         return r0 + r1 * r2;
31         break;
32

```



3. pi-OS system call implementation (TODO in project 3)

Step 1: initialize the SUPERVISOR mode stack

Hint: ARM CPU has USER MODE, SYSTEM MODE, IRQ MODE, FIQ MODE, SUPERVISOR(SVC) MODE. Entering each mode, the CPU has a separate stack. We need to initialize these stacks just at the time when operating system starts up. If you don't initialize the stack for a certain mode, and CPU somehow goes into this mode, invalid stack will cause CPU to go into ABORT MODE. The codes below give you an idea how to initialize stack. **Add code to initialize stack for supervisor mode. You need to know the meaning of these codes.**

```
52  .text
53
54  .globl _start
55 _start:
56 #ifndef USE_RPI_STUB_AT
57     safe_svcmode_maskall r0
58
59     mov r0, #0
60     mcr p15, 0, r0, c12, c0, 0      /* reset VBAR (if changed by u-boot) */
61 #endif
62     cps #0x12                      /* set irq mode */
63     mov sp, #MEM_IRQ_STACK
64
65
66     cps #0x17                      /* set abort mode */
67     mov sp, #MEM_ABORT_STACK
68     cps #0x1B                      /* set "undefined" mode */
69     mov sp, #MEM_ABORT_STACK
70     cps #0x1F                      /* set system mode */
71     mov sp, #MEM_KERNEL_STACK
72     b sysinit
```

Step 2: setup vector table entry for software interrupt

Hint: just like hardware interrupt in Project 2, you also need to put a BRANCH instruction in the vector table entry for software interrupt. **When CPU goes into SUPERVISOR MODE, it will go to this entry address. So your BRANCH instruction will make CPU jump to your first level handler.**


```

68 boolean CInterruptSystem::Initialize (void)
69 {
70     TExceptionTable *pTable = (TExceptionTable *) ARM_EXCEPTION_TABLE_BASE;
71     pTable->IRQ = ARM_OPCODE_BRANCH (ARM_DISTANCE (pTable->IRQ, IRQStub));
72     SyncDataAndInstructionCache ();
73
74     #ifndef USE_RPI_STUB_AT
75     PeripheralEntry ();
76
77     write32 (ARM_IC_FIQ_CONTROL, 0);
78
79     write32 (ARM_IC_DISABLE_IRQS_1, (u32) -1);
80     write32 (ARM_IC_DISABLE_IRQS_2, (u32) -1);
81     write32 (ARM_IC_DISABLE_BASIC_IRQS, (u32) -1);
82
83     PeripheralExit ();
84 #endif
85
86     EnableInterrupts ();
87
88     return TRUE;
89 }

```

Step 3: first level handler

Hint: You first level should **save USER MODE registers, extract system call number, BRANCH to second level handler, restore USER MODE registers.**

You can re-read IRQStub in project 2 and ARM software programming guide. Second level handler would receive four parameters. Three parameters passed from USER MODE are stored in r0, r1, r2, and you should put SVC number in r3. Then second level handler can work correspondingly. When second level handler returns, first level handler would continue, that is to restore USER MODE registers, and switch MODE.

```

68 .globl SWIStub
69 SWIStub:
70     stmfd    sp!, {r1-r12, lr}      /* save r0-r12 and return address */
71
72
73
74     bl SWIHandler
75
76
77
78
79     ldmdfd   sp!, {r1-r12, pc}^     /* restore registers and return */
80
81
82
83
84
85

```

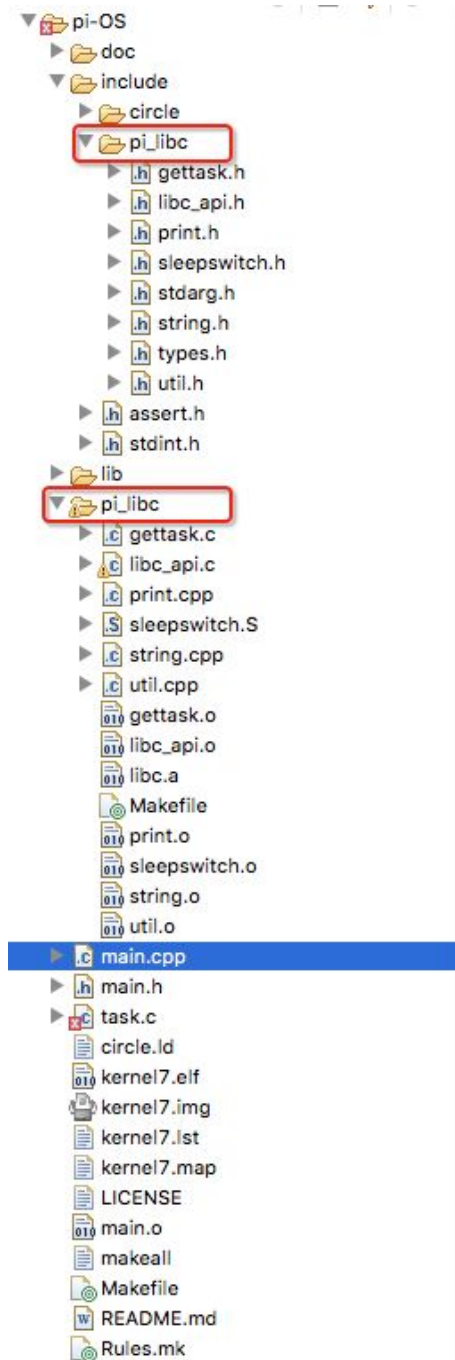
Step 4: second level handler

Hint: you can implement second level handler in C language, and this function receive four parameters(in accordance with r0, r1, r2, r3 passed from first level handler). You can use switch(r3) to branch to different handling functions. You can further call functions in sched.cpp. When second level handler finishes, it will return to first level handler. You can create a new file softwareinterrupt.c in pi-OS/lib folder to put second level handler function.

```
19 int SWIHandler (unsigned r0, unsigned r1, unsigned r2, unsigned r3)
20 {
21     PeripheralExit (); // exit from interrupted peripheral
22     pKernel->write_log("r0=%d,r1=%d,r2=%d,r3=%d", r0, r1, r2, r3);
23
24     switch(r3){
25     case 1:
26         //
27         //
28         //
29         break;
30     case 1000: //printf(char *)
31         return r0 + r1 * r2;
32         break;
33     case 1001:
34         //
35         break;
36     case 1002:
37         //
38         //
39         //
40         return //
41         break;
42     case 1003:
43         //
44         //
45         return //
46 }
```

Step 5: create pi_libc API basing on system call

Hint: Create header file folder for pi C library. You put it under pi-OS/include/ folder, and name it to pi_libc. In pi-OS/include/pi_libc/, you put header files for functions in your C library. Create a C library folder just under pi-OS folder. You can name it to **pi_libc**. In pi-OS/pi_libc/ folder, you will put implementation for pi_libc functions, defined in pi_libc header files. For example, you want to create a **printf** function in your C library. You define it in **pi-OS/include/pi_libc/libc_api.h**:



This is how your new folders should be like.

```
3/pi-OS/include/pi_libc/libc_api.h - Eclipse
Quick Access

libc_api.h
2+ * pi-lib.h
7
8 #ifndef PROJECT3_PI_OS_INCLUDE_PI_LIBC_LIBC_API_H_
9 #define PROJECT3_PI_OS_INCLUDE_PI_LIBC_LIBC_API_H_
10
11
12
13
14
15
16
17
18
19
20
21 int printf(const char * pMessage,...);
22
23
24
25
26
27
28
29
30 #endif /* PROJECT3_PI_OS_INCLUDE_CIRCLE_PI_LIB_H_ */
31
```

And you implement it in pi-OS/pi_libc/libc_api.c:

```
3/pi-OS/pi_libc/libc_api.c - Eclipse

libc_api.c
150 //system call number: 1001;
151 int printf(const char * pMessage,...){
152     va_list var;
153     va_start (var, pMessage);
154     call_printf(pMessage, var);
155     va_end (var);
156 }
157
158
159
```

```

13 void call_printf(const char *pMessage, va_list Args){
14
15     CString Message;
16     Message.FormatV (pMessage, Args);
17
18     _sysCallPrint((const char *)Message);
19
20
21 }
22
23
24 void _sysCallPrint(const char *pMessage){
25     asm("swi 1001");
26 }

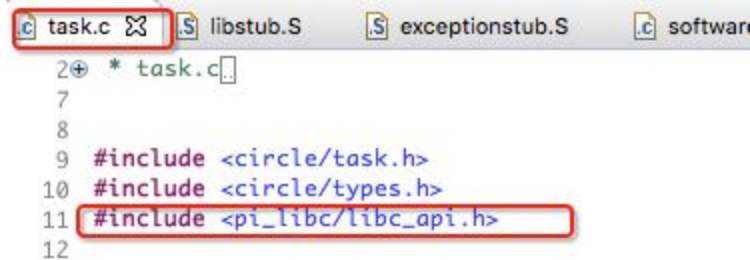
```

In the implementation, it finally make a system call through `asm("swi 1001");`

During lab session, we have talked about how SWI(or SVC) instruction trigger the CPU into SUPERVISOR MODE. So it can serve software interrupt as we've expected.

Step 6: using libc API

In the task.c, you include `pi_libc/libc_api.h`, so you can use the C library functions:

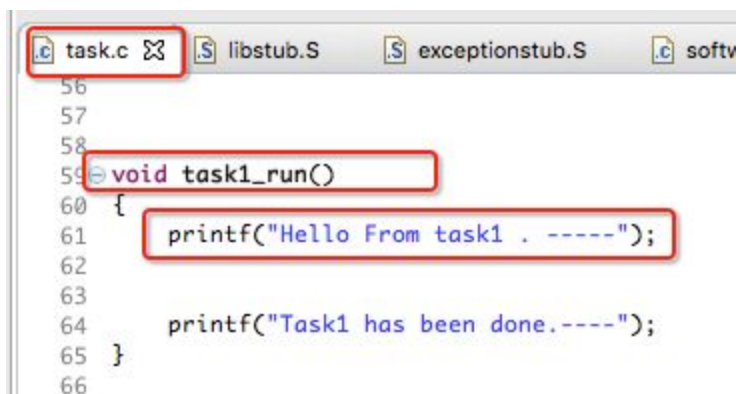


```

2 * task.c
7
8
9 #include <circle/task.h>
10 #include <circle/types.h>
11 #include <pi_libc/libc_api.h>
12

```

For example, in `task1_run()`, you use the `printf` function:



```

56
57
58
59 void task1_run()
60 {
61     printf("Hello From task1 . ----");
62
63
64     printf("Task1 has been done.----");
65 }
66

```

NOTE: Since we're using software interrupt to implement system call, so we won't access system call functions as how we did in project 1 and project 2. So we do not need to pass Task pointer into the task_run function. Your task_run function prototype should look like: `void task1_run();`

4. using kernel api through pi_libc and SVC (TODO in project 3)

We've covered the steps to add SYSTEM CALL mechanism in pi-OS. So now you should be able to replace "system calls" in project 2 with the REAL SYSTEM CALL version. **You need to create these three system calls, using software interrupt.**

- 1) printf(const char * message, ...)
- 2) exit(int errornum)
- 3) sleep(int seconds)

Hint:

- 1) For printf, its system call number is 1001. We've provided you implementation of printf in pi_libc, you only need to add interrupt handlers. You can call kernel.write_log() in your interrupt handler. After handling, CPU would return to the place just after SVC instruction in pi_libc function.
- 2) For exit, its system call number is 1. This function take a ERROR number as exit info, you can ignore its meaning in project 3. When user task call this function, user task want to end executing. And CPU should definitely not go back to this user task after interrupt handling. So after software interrupt, it won't return to user task again. Second level handler should call some scheduling function (you need to create) to remove this user task from task queue. Also, second level handler should find scheduler(kernel) task register pointer, and return this pointer to first level handler. Then the first level handler would load scheduler task's CONTEXT(r0-r13, pc), so it directly goes back to scheduler.
- 3) For sleep, its system call number is 1002. It takes time as parameter. When user task call sleep(seconds), the CPU won't return to user task. Just like exit, it will return to scheduler. Second level handler should call some scheduling function (you need to add) to create a virtual timer for this task. When timer reaches a certain value, its handler will be called. The handler does the same thing as in project 2's real time timer handler. Also, second level handler should find scheduler(kernel) task register pointer, and return this address to first level handler. Then the first level handler would load scheduler task's CONTEXT(r0-r13, pc), so it directly goes back to scheduler.

5. Some suggestion

Read related assembly codes in kernel7.lst. It's essential to look at generated code when mixing C and ARM assembly code. You need to make sure that generated assembly codes can maintain the correct stack environment, maintain correct argument order, and maintain return value.(GCC won't maintain registers used in your own assembly code. So you need to save and restore some registers when writing assembly code).

6. System calls in future projects

Future projects will use system call a lot, since system call is the communicating path linking user task and kernel functions.

File system api

I/O API

....

7. REQUIREMENTS

Do read this document carefully. If you find INTERRUPT mechanism unfamiliar, please read project 2 designing document carefully. Software interrupt serving process is similar to hardware interrupt serving process, so it's necessary for you to read project 2's **interrupt serving process codes several times**. **If you read project 2 document carefully, you'll be able find corresponding codes easily.** You should not start project 3 if you don't fully understand project 2, although their implementations are independent.

If you have trouble mixing C and ARM assembly, please look for answers in references. System call is very important, and later projects will rely on project 3. Devote more time on project 3, it worths.

8. References:

1. Chapter 7 ARM Exceptions.pdf
2. Compiler_user_guide_100748_0606_00_en.pdf
3. ARM software develop official guide.pdf

A very good article on ARM exception system:

<http://www.peter-cockerell.net/aalp/html/ch-7.html>

A very helpful article on system programming:

<http://www.peter-cockerell.net/aalp/html/ch-5.html>

Glib C library:

https://www.gnu.org/software/libc/manual/html_mono/libc.html#Creating-Directories

<http://man7.org/linux/man-pages/man2/intro.2.html>

ARM modes:

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0290g/127695.html>

<http://stackoverflow.com/questions/22295566/how-can-i-put-arm-processor-in-different-modes-using-c-program>

Return from exception:

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0040d/Cacbacic.html>

SVC handler:

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0203j/Cacdfeci.html>

Linux 2.6.35.4 system calls:

<http://syscalls.kernelgrok.com/>

<http://man7.org/linux/man-pages/man2/syscalls.2.html>

A very good article on ARM exception system:

<http://www.peter-cockerell.net/aalp/html/ch-7.html>

Some details:

Can anybody help me with this '^' meaning T_T what is different with

```
stmfd    sp!, {r0-r14}^
```

and

```
stmfd    sp!, {r0-r14}
```

????

From the [ARM manual](#):

^

is an optional suffix. You must not use it in User mode or System mode. It has two purposes: If op is LDM and reglist contains the pc (r15), in addition to the normal multiple register transfer, the SPSR is copied into the CPSR. This is for returning from exception handlers. Use this only from exception modes. Otherwise, data is transferred into or out of the User mode registers instead of the current mode registers.

As none of the routines change the link register, R14, they all return using a simple move from the link register to the PC. We do not bother to use `movs` to restore the flags too, as they are not expected by the main program to be preserved.

If a subroutine calls another one using `bl`, then the link register will be overwritten with the return address for this later call. In order for the earlier routine to return, it must preserve R14 before calling the second routine. As subroutines very often call other routines (i.e. are 'nested'), to an arbitrary depth, some way is needed of saving any number of return addresses. The most common way of doing this is to save the addresses on the stack.

The program fragment below shows how the link register may be saved at the entry to a routine, and restored directly into the PC at the exit. Using this technique, any other registers which have to be preserved by the routine can be saved and restored in the same instructions:

```
;
;subEg. This is an example of using the stack to save
;the return address of a subroutine. In addition, R0,R1
;and R2 are preserved.
;
.subEg
STMFD (sp)!,{R0-R2,link};Save link and R0-R2
... ;Do some processing
...
LDMFD (sp)!,{R0-R2,pc}^ ;Load PC, flags and R0-R2
;
```

The standard forms of `ldm` and `stm` are used, meaning that the stack is a 'full, descending' one. Write-back is enabled on the stack pointer, since it almost always will be for stacking operations, and when the PC is loaded from the stack the flags are restored too, due to the `^` in the instruction.

Note that if the only 'routines' called are `swi` ones, then there is no need to save the link register, R14, on the stack. Although `swi` saves the PC and flags in R14, it is the supervisor mode's version of this register which is used, and the user's one remains intact.

Using registers is just one of the ways in which arguments and results can be passed between caller and callee. Other methods include using fixed memory areas and the stack. Each method has its own advantages and drawbacks. These are described in the next few sections.

Register parameters

On a machine like the ARM, using the registers for the communication of arguments and results is the obvious choice. Registers are fairly plentiful (13 left after the PC, link and stack pointer have been reserved), and access to them is rapid. Remember that before the ARM can perform any data processing instructions, the operands must be loaded into registers. It makes sense then to ensure that they are already in place when the routine is called.

The operating system routines that we use in the examples use the registers for parameter passing. In general, registers which are not used to pass results back are preserved during the routine, i.e. their values are unaltered when control passes back to the caller. This is a policy you should consider using when writing your own routines. If the procedure itself preserves and restores the registers, there is no need for the caller to do so every time it uses the routine.

The main drawback of register parameters is that they can only conveniently be used to hold objects up to the size of a word - 32-bits or four bytes. This is fine when the data consists of single characters (such as the result of `swi readc`) and integers. However, larger objects such as strings of characters or arrays of numbers cannot use registers directly.