

## Homework 7

### Objectives of the homework:

- Introduction to system calls and user space (inspired by an experimental class at Stanford University and the University of Maine).

### Required materials:

- Raspberry Pi v3B and serial connector
- And/or QEMU

### Preliminary installations:

- See all previous homework
- You may want to revisit the lectures about System Calls.

### External useful documentation:

- 1 ARM architecture reference manual [developer.arm.com ARM7](https://developer.arm.com/ARM7)
- 2 OS Three Easy Pieces [Limited Direct Execution](#)

### Overall presentation.

In this homework, you will build your own system calls for the rPi. First, take some time to read the corresponding chapter in the book OS Three Easy Pieces. Many answers are there!

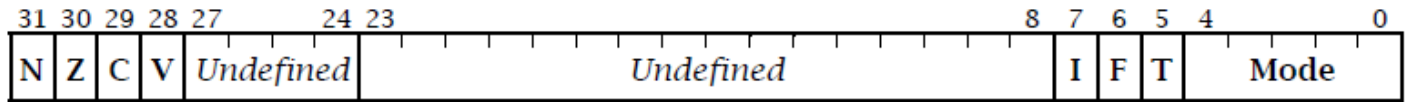
### ARM Operating modes (this has already been discussed in a previous lab).

You can read, in the ARM Architecture manual, that ARM has several operating modes:

Mode	Privileged	Purpose
User	No	Normal operating mode for most programs (tasks)
Fast Interrupt (FIQ)	Yes	Used to handle a high-priority (fast) interrupt
Interrupt (IRQ)	Yes	Used to handle a low-priority (normal) interrupt
Supervisor	Yes	Used when the processor is reset, and to handle the software interrupt instruction swi
Abort	Yes	Used to handle memory access violations
Undefined	Yes	Used to handle undefined or unimplemented instructions
System	Yes	Uses the same registers as User mode

Modes change either because of external interrupt processing (IRQ, FIQ), internal processing (executing an undefined instruction), or by software intentionally executing an instruction that generates an exception. Most applications execute in user mode. In that mode, the program is unable to access some protected system resources (like uart routines). All modes – except user-mode – have full access to system resources and can change mode freely. To access other modes, a program in user mode will issue a SYSCALL.

CPSR.



The Current Program Status Register (CPSR) is used to store condition code flags, interrupt disable bits, the current processor mode, and other status and control information.

## Registers and bank registers.

Each processor mode has its own R13 and R14 registers. This allows each mode to maintain its own stack pointer and return address. In addition, the Fast Interrupt (FIQ) mode has additional registers: R8–R12. This means that when the ARM processor switches into FIQ mode, the software does not need to save the normal R8–R12 registers, as FIQ mode has its own set that can be modified. These additional registers are called **bank registers**.

Note: each mode has its own stack which must be initialized at boot time (see the program **BOOT.S**).

## Software Interrupt.

A *software interrupt* is a type of exception that is initiated entirely by software. On the ARM processor, the relevant instruction that does this is `svc` (previous version used the `swi` instruction, which is now deprecated). When this instruction is executed, it causes the processor to switch into **Supervisor mode** and branch to the relevant exception vector address, `0x00000008` which contains the address of a routine that calls the `svc` handler.

In other words, `svc` causes an *intentional* exception that is foreseen by the program. To the ARM processor, `svc` is just another type of exception. When the processor executes this instruction, it:

1. Copies the address of the next instruction following the **svc** into the **LR\_svc** (R14\_svc) register.
2. This return address is actually  $PC - 4$ ; the **svc** instruction can be found at  $PC - 8$ ,
3. Copies the CPSR into **SPSR\_svc** (the Supervisor mode SPSR),
4. Sets the CPSR mode bits to Supervisor mode. This has the effect of “swapping in” **R13\_svc** and **R14\_svc** and “swapping out” the previously visible R13 and R14,
5. Enforces ARM state by setting bit 5 (the T bit) of CPSR to zero,
6. Disables normal interrupts by setting bit 7 (the I bit) of CPSR to one. This means that normal interrupts cannot cause exceptions during the **svc** call unless bit 7 is later set to zero in the exception handler’s code. Fast interrupts are *not* disabled and can still occur;
7. Loads the address of the exception vector, 0x00000008, into the Program Counter PC.

Once the software interrupt handler has finished its task, it returns control to the calling program by:

1. Moving the contents of register **LR\_svc** (= R14\_svc) into PC, and
2. Copying **SPSR\_svc** back to CPSR.

The following single instruction performs both of these steps:

```
movs pc, lr ; Copy current LR to PC and copy current SPSR to CPSR
```

Note that the instruction is **movs**, not **mov**: the **movs** instruction automatically copies SPSR to CPSR, but *only* when the destination register is PC (=R15) and the instruction is executed in a privileged mode.

## Homework 7

Example of `svc_handler`:

```
SVC:
// This executes in SVC mode automatically
    push    {r4-r12, lr}          ;@ save registers
    bl      svc_handler
    pop     {r4-r12, lr}          ;@ restore registers
    movs    pc, lr                ;@ jump back to caller
```

The `svc_handler` (a C program) will handle the system call. But how does it know what to do?

We saw in class that the register `r7` should contain a value that identifies the system call. We will do something simpler (because it is tricky to set the register `r7` from C). We will use register **`r3`** to identify the system call and registers `r0-r2` will be used to pass parameters if any (not modified between USER mode and SVC mode):

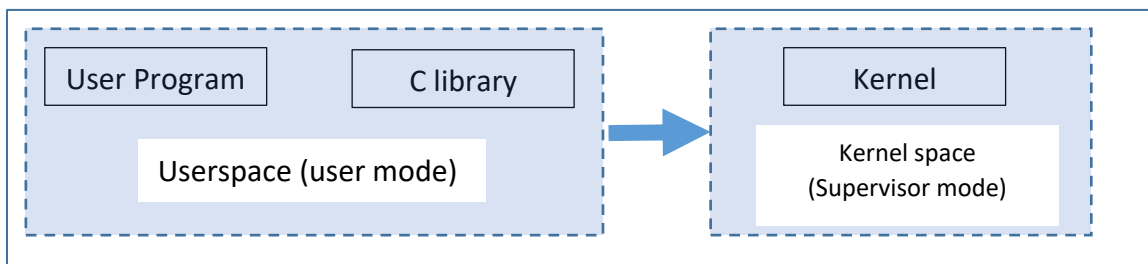
- `r3` will contain the type of action the SYSCALL should perform
- `r0-r2` will contain possible parameters
- At the end, we will return the result in `r0`

The routine `callSVC` is very simple (see `pi_utils.s`). Do not forget to have `SPuser` initialized (I did not do that in an early version, and it took a long time to figure out what was going wrong):

```
;@-----
;@ Call SVC (from user mode).
;@ Input r0,r1,r2,r3 : SVC parameters
;@ Output r0,r1,r2,r3 : SVC parameters
;@-----
callSVC:
    push    {lr}
    svc     #0
    pop     {pc}
```

### From user-space to kernel-space and back.

We saw that user programs run in user mode, and the kernel runs in a privileged mode (supervisor mode). We also saw that the processor will switch to supervisor mode when a user makes a system call. But how does a user program make a system call? Of course, it could populate the appropriate registers, indicate the system call number, and invoke the `callSVC()` function, but this is tedious. Another possibility is to use a C library that will wrap the system call, populating the registers with the appropriate values. So the path will be:



## Homework 7

### Practice 1: Developing a kernel log.

The kernel keeps its logs in a ring buffer. That is an array in memory that is managed as a circular buffer. The main reasons are that the log must be available as soon as the time the system boots and must consume as little time as possible (so writing on the disk is a no-no). You will need to write the routines to manage the circular buffer.

#### How does it work?

The kernel log is an **array of char** containing only ASCII characters, the character '\0' marks the end of a string. There is only one index that points to the next position to write to. After writing a string, the index is moved accordingly. The log is said circular because the character after the last one in the array is the first character in the array. Example with an array of 10 characters (the index is in red):

Starting position:

0	1	2	3	4	5	6	7	8	9
\0	\0	\0	\0	\0	\0	\0	\0	\0	\0

Log "HELLO"

0	1	2	3	4	5	6	7	8	9
H	E	L	L	O	\0	\0	\0	\0	\0

Log "CS3281"

0	1	2	3	4	5	6	7	8	9
8	1	\0	L	O	\0	C	S	3	2

So the messages are overriding each other but if the buffer is large enough, it can keep the last messages that have been added (which should be retrieved backward from the last one to the first one). In the example above, dumping the log file would display:

0- LO  
1- CS3281

**Note 1:** you should number the messages from 0 (oldest message) to the last one inserted.

**Note 2:** if the buffer is larger, then it would contain:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
H	E	L	L	O	\0	C	S	3	2	8	1	\0	\0	\0	\0	\0	\0	\0	\0

In this example, dumping the log file would display:

0- HELLO  
1- CS3281

### Practice 2: Writing your own system calls.

#### The shell

Once the kernel boots, real-life operating systems will wait for a user to log in. Then will create a process for that user which will execute a shell to read user commands. Our kernel will not implement a login mechanism. We will directly switch to user mode and execute system calls from a rudimentary shell that is provided to you. Since the shell is running in user-mode, it should not use any of the UART routines. But I did not have

## Homework 7

time to rewrite-it so you have it “as-is”. But your routines in the `pi_vlibc` library should absolutely not use these UART routines.

This shell has 2 variables (one integer and one string) that act as a buffer (since we don’t want the OS to write on the screen and intermingle with the shell).

### The library `pi_vlibc.c`

You can see the list of the C library routines (in `pi_vlibc.c`) you need to implement by typing `HELP` in the shell. One example is given: `getRND()`. This routine will perform a system call to get a random number but will also adjust the returned value to be less than a value. Please note that: because the code executes in **user mode**, you can’t use any UART calls in the system library (because only O.S. should access it). The call to the O.S. is done using `callSVC`, the first 3 parameters are null (they are useless) and `r3` equals `SYS_RND` which will tell the O.S. what to do.

Most of them are simple: some verification and setting registers `r0/r1/r2/r3` to the right values, then execute a `callSVC`.

### The system calls library: `pi_syscalls.c`

A `callSVC` will execute a `SVC #0` which will trigger the software interrupt (and a change to supervisor mode). This will trigger the `syscall_handler` routine. This one receives all the information from registers `r0/r1/r2/r3`. The value in register `r3` determines which system call the handler will execute. The system call `sys_rnd()` is given to you as an example. They are all quite simple – all of them should start with logging in the kernel log a message that has:

[name of system call] value of registers passed as parameter.

Here are the system calls you must implement:

1. `sys_log`: (action). This will either reset (action == 0) or dump (action == 1) the content of the kernel log. This kernel log is a circular buffer used to store messages as the kernel executes tasks.
2. `sys_read`: (file\_descriptor, address where to store characters read, number of characters) will read a certain number of characters from the file\_descriptor (only from the keyboard at this time) and will store them at the provided address.
3. `sys_write`: (file\_descriptor, address of string) will write the characters in the string until the null terminator.
4. `sys_panic`: (address of string to be printed).

Of course, since your system calls execute in supervisor mode, you can access the hardware directly (e.g. use `uart_getByte()` or `uart_readByte()` ).

### Reading from `SYSIN`

The result of implementing reading from `SYSIN` with `uart_getbyte()` will mess up the shell display. So you should use `uart_getbyteFAKE()` which is going to return a character (as if a user typed something).

The information will flow as follows:

## Homework 7

### shell

IOVariable (String): 

```
readStr (keyboard, &IOVariable, 80)
```

### pi\_vlibc

```
readStr (fd, *buf, count)
```

### pi\_syscalls

```
sys_read (r0, r1, r2)
```

The shell will pass to the library the address of the temporary IOVariable that will hold the characters to be read, the readStr will call sys\_read passing the address of that variable and sys\_read will fill it in.

### panic

It is very difficult to recover after some errors. It is a good idea to have a routine that:

- Prints a message (which routine calls **panic**).
- Dumps the content of all registers on screen.
- Dumps the content of the CPSR register on the screen.
- Dumps a certain number of words from the stack (say 4 but you can choose something different) on screen.
- And stops execution (halt – defined in boot.s).

The corresponding routines are available in debug.c.

## Homework 7

### Example of execution:

```
lab07:run
interrupt_init
[pi_kernel version 0.1]Starting
USR>abcd
Unknown command: ABCD
Type HELP.
USR>help
The prompt of the mini-shell indicates the CPSR mode.
Available commands are:
READ n    to read n characters from the keyboard and
           store them into a temporary variable.
ECHO      to write the content of the temporary variable to the screen
LOGR      to reset the kernel log.
LOGD      to display the kernel log.
DIE        to enter panic routine and stop.
RND n     to obtain a random number < n and
           store it into a temporary variable.
RNDSHOW   to display the random number.
HELP      to display this message
HELP x    to display specific help about a command.
USR>rnd 1000
USR>rndshow
|838|
USR>read 10
USR>echo
|yes zGqnye|
USR>logd
-- Dumping kernel log --
0- [SVC_HANDLER] Mode = SVC r0=0x0 r1=0x0 r2=0x0 r3=0x5
1- [SYS_RND]
2- [SVC_HANDLER] Mode = SVC r0=0x1 r1=0xE038 r2=0x0 r3=0x3
3- [SYS_WRITE] r0 = 0x1 r1 = 0xE038
4- [SVC_HANDLER] Mode = SVC r0=0x0 r1=0xE038 r2=0xA r3=0x2
5- [SYS_READ] r0 = 0x0 r1 = 0xE038 r2 = 0xA
6- [SVC_HANDLER] Mode = SVC r0=0x1 r1=0xE038 r2=0x0 r3=0x3
7- [SYS_WRITE] r0 = 0x1 r1 = 0xE038
8- [SVC_HANDLER] Mode = SVC r0=0x1 r1=0x0 r2=0x0 r3=0x4
9- [SYS_LOG] r0 = 0x1
-- End of kernel log --
USR>die
Panic mode created by from shell

Dump Register - PANIC
r0 = 0x0      r1 = 0xA      r2 = 0xC8D8      r3 = 0x0
r4 = 0x0      r5 = 0xA      r6 = 0xC8D8      r7 = 0x0
r8 = 0x3E     r9 = 0x10000   r10= 0x0       r11= 0x0
r12= 0x0      r13= 0x7DF8   r14= 0xA474     r15= 0xB360
CPSR = 0x600001D3 IRQ[1] - mode:SVC
Dump Stack - PANIC
SP = 0x7DF8   dumping 4 words
[0x7DF8] = 0x9 [0x7DF9] = 0x5B [0x7DFA] = 0x30 [0x7DFB] = 0x78
[0x7DFC] = 0x37 [0x7DFD] = 0x44 [0x7DFE] = 0x46 [0x7DFF] = 0x45
[0x7E00] = 0x5D [0x7E01] = 0x20 [0x7E02] = 0x3D [0x7E03] = 0x20
[0x7E04] = 0x30 [0x7E05] = 0x78 [0x7E06] = 0x37 [0x7E07] = 0x37
```

A random number (< 1000) is returned from the O.S. and stored into the RNDVariable (int) in the shell.

The RNDvariable in the shell is copied to the IOvariable (char) and library call writeStr is executed.

10 characters are returned from the O.S. (uart\_getbyteFAKE) and stored into the IOVariable (char) in the shell.

The library call writeStr is executed.

-o-o-o-o-o-o End of Homework 7 -o-o-o-o-o-o