Lab 15: System calls (II)

This practice is a continuation of past practice, therefore their goals are the same. The handler Mimos_v3.handler will be taken as starting point, where the four system calls *get_version*, *print_char*, *get_time* and *wait_time* are implemented. We will implement a new system call, *read_char* and *print_char* will be modified.

Material

- PCSIM Simulator version used in the previous practice.
- The preliminary version of the handler *MIMOSv3.handler*.
- Test file: *Usuario.s*

Processes in MiMoS

Remember that this version of the handler provides multiprocessing capacity, so there is also defined an **idle process** as follows:

```
void_process: # System void process
             b void_process
```

This process is always active and its context is minimal, because no processor registers are used and, when running, the program counter constantly points to the direction void_process.

The handler code is always invoked through an exception, so it should be determined at the end of its execution to which instruction has to return.

Remember that at the end of the exception handler code (tag retexc) it is the code that performs the **process management** and leaves the returning address on *$k0*. There are only two options: If the user process is ready, then it enters execution; otherwise it is the idle process:

```
If (state == ready)
   $k0 = return to user process address
else
   $k0 = void_process
end if
```

Also remember that the **context change** is not necessary, because you only have to maintain the context of the main process.

Task 1. Handler version 3 (MiMoS v.3). Function read_char.

Version 3 of MiMoS is presented with the system calls *get_version get_time*, *print_char* and *wait_time*. System call *read_char* that reads a character on the keyboard, shown on **table 1**, is the one that has to be implemented now, and in the following section we will implement a new version of system call *print_char*.

The system call *read_char* will put the user process in standby, until the keyboard is ready to provide a character. The keyboard interrupt will do the change of user process state, when the user presses a key, and the character read will be returned through $a0 register.

Function	Code	Arguments	Results
`get_version`	*$v0* = 90		Version number (in *$v0*)
`get_time`	*$v0* = 91		Time in seconds (in *$v0*)
`wait_time`	*$v0* = 92	$a0 = time in seconds	
`print_char`	*$v0* = 11	$a0 = character to print	
`read_char`	*$v0* = 12		$a0 = read character

Table 1 : Services to implement in MiMoS v.3. Get_version, get_time, print_char and wait_time services are already implemented in the handler. The so-called read_char must be implemented.

The keyboard must have its interrupt enabled only from the moment in which the program calls ***read_char*** until when the peripheral is available. To simplify the management of the keyboard, its interrupt line *int0* * will be unmasked permanently on the coprocessor state registry and enabling/disabling operations will be done on the keyboard interface.

The sequence of events that occur during waiting for the return of function *read_char* is shown in Figure 1. As in the former lab, the idea is to change the status of the process that calls function *read_char* from the value READY to the value `WAITING_READ`. With this action the handler does not return to the parent process, but to the idle process. The keyboard interrupt routine will be in charge of restoring the state to the value READY, which will cause the return to the main process.

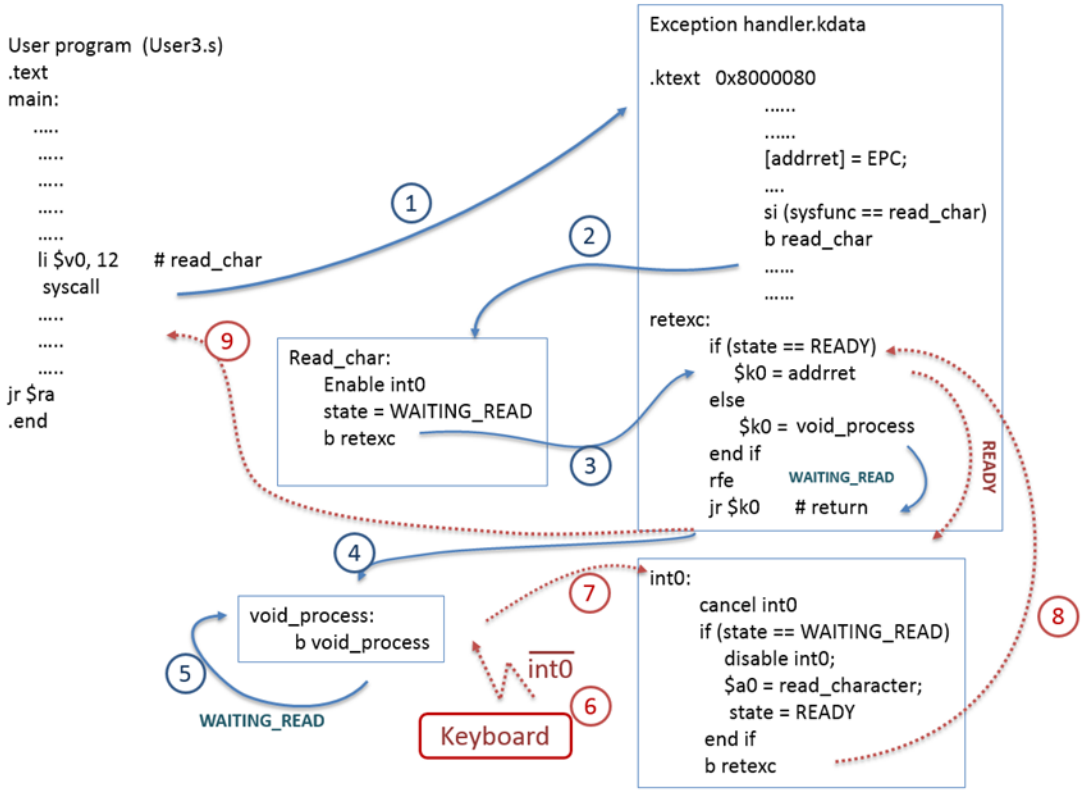

Figure 1 – Process switching on *read_char* calling.

Perform the following changes to *MiMoSv3.handler*:

► Implement *read_char*. The handler has to put the user process in state `WAITING_READ` and enable the interruption in the keyboard interface. To do so you have to define constant `WAITING_READ` in .kdata where the other state constants are defined.

```
read_char:
            Enable keyboard interrupt;
            state = WAITING_READ;
            b retexc
```

Question 1. Enter the code corresponding to the system call *read_char*.

read_char:

► Write the keyboard interrupt handling code (label `int0`). It should only be applied if the user process is on `WAITING_READ` and it consists of leaving the character read in *$a0*, cancelling and disabling the keyboard interrupt and putting the user process on READY state.

Note that this service is limited to read the keyboard character without printing it on the screen.

int0:
    If (state == WAITING_READ)
      $a0 = read character;
      Cancel and suppress keyboard interrupt;
      State = READY *;*
    end if
    b retexc

Question 2. Enter the code corresponding to the keyboard interrupt, from tag *int0*.

int0:

► Set the mask of interrupts in the handler startup section, line INT0 * enabled (in addition to the clock).

Type the mask value in hexadecimal.

► Test handler using the program *usuario.s* that does the following operations:

Inital welcome;
***get_time** call;*
Write to the console the current time;
***read_char** call;*
Write in the console read character;
***get_time** call;*
Write to the console the current time;

Task 2. Handler version 3 (MiMoS v.3). Function print_char.

We are going to modify the character printing service *print_char* (see **Table 2**) to be synchronized by interrupt, now it is implemented by polling. The mechanism will be the same as on *read_char*: the console interrupt must only be enabled from the moment when the program calls function ***print_char*** until the time when the peripheral is ready, leaving the process in *WAITING_WRITE.* And the console interrupt will do the process state change and it will do the character printing.

Function	Code	Arguments	Results
`print_char`	*$v0* = 11	$a0 = character to print	

Table 2 : New service to implement print_char.

Perform the following list of activities:

► Implement the code of *print_char* . The new handler has to put the user process on *WAITING_WRITE* state and enable the interruption in the console interface.

```
print_char:
        enable console interrupt;
        State = WAITING_WRITE;
        b retexc
```

Question 3

Write the code for the system call *print_char.*

print_char:

► Write the console interrupt (label `int1`) handler code, which should only be applied if the process is found on state `WAITING_WRITE` . It has to copy *$a0* to the data register, to cancel and to disable console interrupt and to put the user process on `READY` state.

```
int1:
        If (state == WAITING_WRITE)
                character to write = $a0;
                cancel and disable console interrupt;
                State = READY;
        end if
        b retexc
```

Question 4

Enter the code corresponding to the console interrupt, from label int1.

int1:

► Set the interrupt mask on the handler startup section. To test the new handling any previous user program can be used, because all of them use function `print_char`

Type the mask value in hexadecimal.

FINAL REMARK

Apart from considering that we have worked on a simulator instead of a real system, the most significant MiMoS restrictions are the following:

- PCSpim doesn't simulates processor system mode, so the program user can, for example, execute privileged instructions and perform direct access to the peripheral interfaces without calling to system code (MiMoS).
- PCSpim doesn't simulates MIPS virtual memory management unit, this is why MiMoS cannot implement memory protection mechanisms.
- MiMoS is written in assembly language, so it would be very difficult to provide it with features implemented on nowadays operating systems that require careful programming and complex data structures.

Relevant operating systems are written with similar limits. CP-M and DOS were two operating systems for microprocessor that were developed with methods similar to the MiMoS. Certainly, the work was carried out on a team and using development tools designed to the job.