ESTRUCTURA DE COMPUTADORES

Lab 14: System calls (I)

Introduction and objectives

The MIPS instruction set provides primitives to implement an operating system: operation modes, exceptions, coprocessor and others. To practice with basic input/output methods, we have defined a rudimentary operating system called MiMoS (*Mips Monitor System*). The structure of this handler is similar to the one completed in lab 13. That handler only had the ability to deal with interrupts INT0*, INT1* and INT2*. This new handler adds the capability of managing system call made with *syscall*. It is also able to perform basic concurrency.

This lab objectives are:

- To implement new functions accessible from user programs as system calls using *syscall*. A first handler version is provided named *MIMOSv0.handler*, that contains the implementation of system calls *get_version* and *print_char*.
- To understand how operating systems deal with concurrency, relying on a basic process management mechanism. Some system calls will leave the user process suspended and will switch the CPU to a void process. The user process will return to execution through an interrupt.

Material

- Simulator PCSIM_ES the same one used in lab 13.
- Provided starting handler version *MIMOSv0.handler*.
- Test user programs: *User0.s*, *User1.s* and *User2.s*. This files will no be changed and they will be used as indicated to test de successive handler versions. *Usuario0.s* uses the system calls already implemented on *MIMOSv0.handler*. *User1.s* will test *MIVOSv1.handler*, and so on.
- Appendixes: MIPS system calls, coprocessor state register bits and peripheral interfaces (keyboard, console and clock).

The simulator settings have to match the configuration on Figure 1, “Syscall Exception” have to be set.

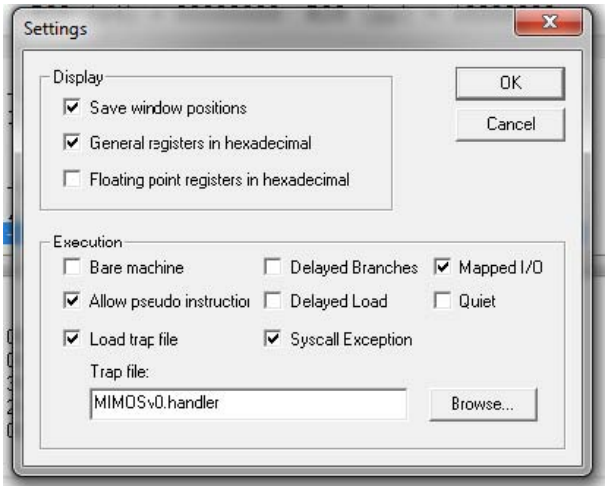

Figure 1: **PCSpim settings to activate MIMOSv0.handler and syscall exceptions.**

Remember what we saw in the former lab: We will use two files:

- The handler or *Trap file*, with extension *.handler*, defines exception hadler segments *.kdata* and *.ktext* and a *.text* fragment for starting and ending codes.

- The user program, with extension *.s* that contains the remaining content of segments *.data* and *.text*.

Every time that a user program is open (*File>Open*) or reloaded (*Simulator>Reload*), the simulator will load also the file indicated as *Trap File* on the settings window shown in figure 1.

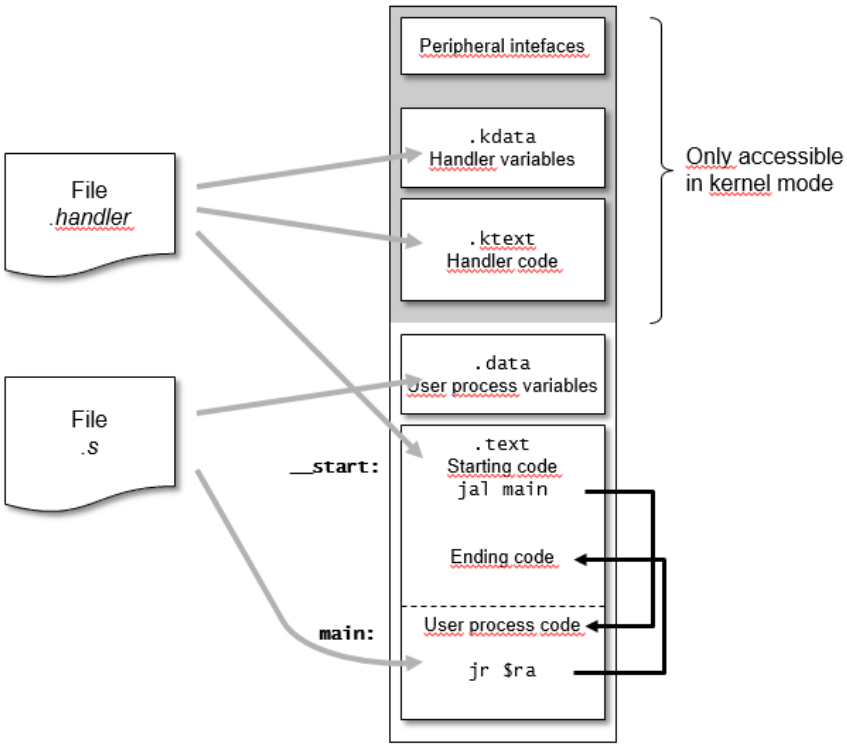

Figure 2: **File mapping on memory of code files.**

The system initialization requires three steps:

1. Setting the available peripherals, enabling or disabling interrupts on their respective interphases.
2. Setting the exception coprocessor state register: interrupts mask, processor execution mode and enable/disable global interrupts.
3. Transferring control to the user program.

Step 3 is done by `jal main`, so the user program will have an entry point marked by label `main`. To end execution the user program will execute `jr $ra` (Figure 2).

Structure of MiMoS handler

Figure 3 describes the handler decomposed into blocks, they appear in the same order as on the source file *MiMoSv0.handler* and with the same labels. Only the *syscall* handling has been added. The handler only includes the implementation of two system calls *print_char* and *get_version*. Along the lab session new functionality will be added.

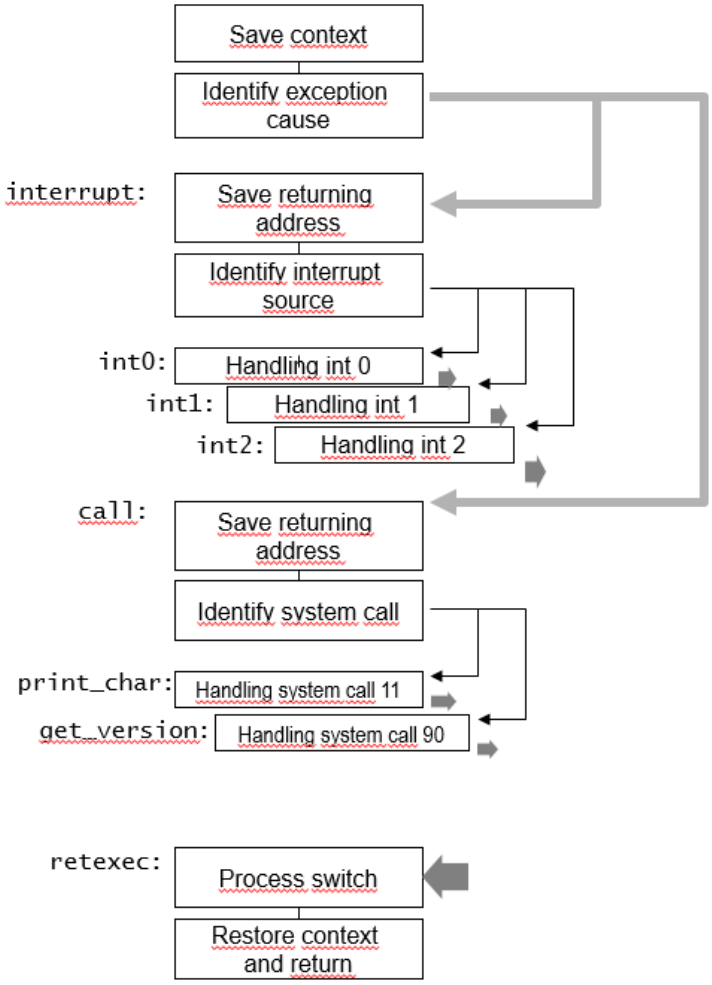

Figure 3: **Estructura del manejador MiMoS. Los símbolos ➡ representan saltos a la etiqueta `retexc`.**

The handler, after saving the context, only identifies as a cause of exception a peripheral interrupt and a system call. In both cases, and after storing the return address, there is a section of identification (interrupt line or function) that jumps to the label where the assigned handling code

starts. Every handling section has to end with `b retexc` instruction to jump into switching and context restoring section.

The returning address is stored in a different way, as explained in theory, for simplicity when choosing between void and main process. Only when the interrupted process is the main one the returning address is stored, if it was the void process the returning address is known, since it has only one instruction.

On *MiMoSv0.handler*, all interrupt handling blocks have to be completed. Global interrupts are enabled, but both the peripherals and the interrupt lines are masked. Functions accessed by *syscall* (except `get_version` and `print_char`) have to be implemented, although the labels and the final handling jump are already defined. The remaining handler sections are already done.

Task 1. MiMoS v.0: Functions get_version and print_char.

MiMoSv0.handler only has two system calls implemented: `get_version` and `print_char` (see table 1).

Function	Code	Arguments	Results
`get_version`	*$v0* = 90		*$v0* = version number
`print_char`	*$v0* = 11	*$a0* = character	Print character on the console

Table 1. System calls implemented on *MiMoS v.0*

► Look at the MiMoS v.0 code and check that it conforms to the structure of Figure 3. Look for the implementation of the two services, you'll see that `get_version` just returns $v0 = 0 and that `print_char` writes a character to the console synchronizing by polling. Along the lab session the handler will be completed developing new versions; and in every version you will have to update `get_version`.

►Look at the starting code. As *MiMoS v.0* doesn't handle any interrupt, notice that any of the interrupts to deal with (corresponding to clock, keyboard and console) are disabled twice: on the interfaces (described on appendix 3) and on the exception coprocessor state register (described on appendix 2).

► In order to check the handler we need a user program that uses the functions that the handler provides. Use *User0.s* to check *MiMoSv0.handler*. This user program uses the functions on Table 1 and when executed it has to show on the console the following text:

```
MiMoS v.0
1
2
...
```

► Check the whole system on the simulator. Don't forget to adjust the simulator settings before loading the user program (Figure 1). You can stop the simulation by clicking the PCSpim menu bar on *Simulator>Break* or pressing [*Control-C*].

Question 1. Stop the simulation while running *Usuario0.s*. (Before clicing on "Si" button look at "Execution paused by the user at PC = ...", and check if *PC* points to a user program instruction, that means an address of type "0x0040*nnnn*". If it points to an address of type "0x8000*nnnn*" press "No" and try again).

►What is the value of coprocessor `Status` (register $12) that appear on the top simulator panel?

►In what mode is the processor running? Are the interrupts enabled?

►What is the value of the interrupt mask bits?

► Indicate with what instructions on the *MiMoSv0.handler* starting code are the interrupts disabled for the keyboard, clock and console.

► Indicate how the coprocessor state register is initialized.

From now, several *MiMoS* versions will be succesively developed that will include new functions. **A new version will be created always from the previous one, create a copy of the handler code file renamed with the new version number**.

Task 2. MiMoS v.1: Function get_time.

Now we are going to add the *get_time* service that will allow knowing the time in seconds from starting user program execution. The function is described on Table 2, together with the already implemented services.

Function	Code	Arguments	Results
`get_version`	*$v0 = 90*		$v0 = version number
`print_char`	*$v0 = 11*	*$a0* = carácter	Writes a character on the console
`get_time`	*$v0 = 91*		$v0 = time in seconds

Table 2: **Services implemented on *MiMoS v.1***

► Make a copy of file *MiMoSv0.handler* and name it *MiMoSv1.handler* and work on the new file. Modify service `get_version` in order to get value 1.

► On the handler data section ("Clock variables" part) add a new variable named *seconds* of word type and initialized to 0.

► On the interrupt handling part, from label `int2`, you have to write the clock interrupt handling code that will be executed every second. It has to cancel the interrupt, writing a 0 at bit EOC (R bit) on the clock interface (described on appendix 3) and then it will increment variable *seconds* into 1. This is the pseudocode:

```
int2: segundos = segundos + 1;
      cancelar interrupción;
      b retexc
```

Question 2. Write the code to handle the clock interrupt.

```
int2:
```

Question 3. Modify the starting code in such a way that the clock interrupt become enabled, both on the clock interface and on the coprocessor state register.

► Now the code corresponding to the system call part has to be added from label `get_time`. The handling is almost the same as on `get_version`, but you have to load into *$v0* the value of variable *seconds* instead of a constant.

```
get_time:
        $v0 = seconds;
        b retexc
```

Question 4. Write the code of function *get_time*.

get_time:

► Update the simulator settings window so the new handler be loaded and check the new function using *User1.s* , it does basically the following:

Initial wellcome;
Repeat
    compute
    call `get_time`
    write actual time

Question 5. *User0.s* can be executed correctly with handler *MiMoSv.1*? *User1.s* can be executed correctly with handler *MiMoS v.0*? Explain your answer.

MiMoS processes

We can abstract the running user program and name it main process. So far, this process was always active, but henceforth we will have to consider other states. The state of the main process is stored on the handle variable *state*. We define two states by two respective constants.

```
## Main program states
          READY = 0
          WAITING = 1

state: .word READY # Main process initial state
```

There is also defined a void process, to which the CPU switches when the main process enters the *waiting* state. The void process code is the following:

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

This process is always ready and its context is made only by its starting address (known by the handler), it has only one instruction that jumps to itself.

As the code of the handler is always invoked through an exception, at the end of its execution it needs to know to what instruction it has to return. At the end of the exception handler code (label `retexc`) is the **process switch** code and the code that puts the returning address on $k0. There are only two options: If the process user is READY, then it enters execution; in any other case the process that enters execution is the void process:

if (`state == READY`*)*
    $k0 = main process returning address
else
    $k0 = `void_process`
end if

The change of context is not necessary in this case, because you only have to maintain the context of the main process, since the void process doesn't need to save any context. During the handler execution the PC content is stored on variable *addrret* and the three general purpose registers on the CPU ($at, $t0, and $t1) are saved on *savereg* so the handler can use them. If you need any more registers you can extend *savereg* and modify the save and restore context code so more registers are preserved.

Task 3. MiMoS v.2: Function wait_time.

Now we are going to add to MiMoS the `wait_time` function, described on Table 3. This system call allows the main process to suspend itself during a time interval, it is similar to UNIX system call `sleep()`.

Function	Code	Arguments	Results
`wait_time`	*$v0* = 92	$a0 = time in seconds	Process waiting along $a0 secs

Table 1: New waiting service to be added to MiMoS v.2

When the main process calls `wait_time` it gets suspended, so the `wait_time` code will change the main process state to WAITING and the CPU has to be allocated to another process. Therefore, the handler will not return to the main process but to the void process. The clock interrupt routine will be responsible for restoring the READY state to the main process when the time specified when calling `wait_time` has elapsed.

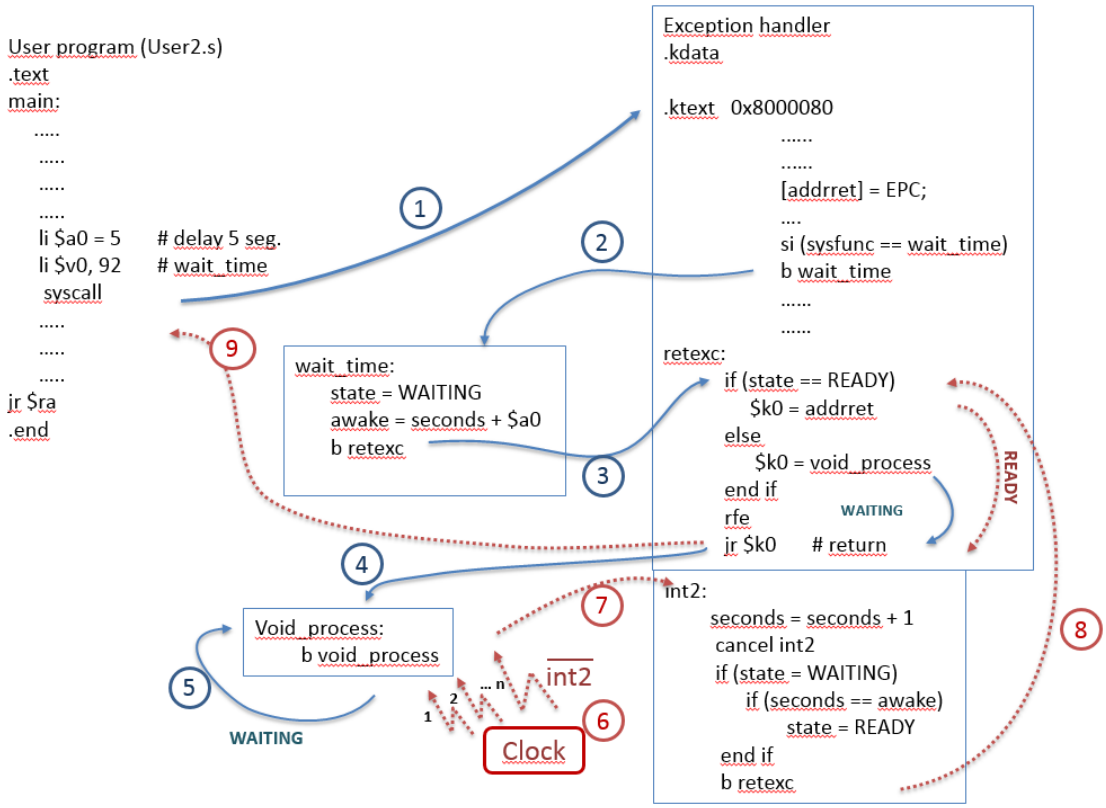

Figure 4. Process switch when calling *wait_time*.

The events that happen are the following:

1. Call to sytem call number 92 (*wait_time*).
2. The exception handler stores the returning address of the main process on variable *addrret* and checks that the funciont is 92, then it jumps to *wait_time* code.
3. *wait_time* changes the state of the main process to WAITING and then jump to *retexc* to return.
4. The returning code checks that the main process IS NOT on the READY state and, therefore, it return to the void process.
5. The void process is running (doing nothing but jumping to its only instruction).
6. Several clock interrupts happen, on every interrupt variable seconds is incremented by 1.
7. When the "n" interrupts happen ("n" is the value passed to `wait_time`) the exception code notices that the waiting has ended, so it changes the main process state to READY and jumps to *retexc* to return.
8. The returning code notices that the main process is READY, so the returning address is taken from *addrret.*
9. The CPU is allocated back to the main process.

► Make a copy of *MiMoSv1.handler* file and name it *MiMoSv2.handler*, work on the new file. Modify service `get_version` to give value 2.

► The operation of `wait_time` will consist of leaving the main process in WAITING state and compute and store in a new variable *awak*e, at what time (actual value of *seconds* variable plus the waiting time specified) the main process will resume execution.

```
wait_time:
        state =WAITING;
        awake = seconds + $a0;
        b retexc
```

Question 6. Write the implemented code for *wait_time*.

wait_time:

► The clock handling code (in label `int2`) has to increment variable seconds into 1, and it has check if the main process state is WAITING. In this case, if *seconds* is equal to *awake*, change the main process state into READY.

```
int2:
        seconds = seconds + 1;
        cancel interrupt;
        if (state == WAITING)
                if (seconds == awake)
                    state = READY;
                end if
        end if
        b retexc
```

Question 7. Write the clock interrupt handling code added from label int2.

```
int2:
```

► Check the new handler updating the settings window and loading *User2.s* into the simulator. This is a test program that basically does the following:

Initial wellcome;
Repeat
    call `get_time`*;*
    write actual time;
    call `wait_time` *(5 seconds);*

Question 8. Stop *User2.s* just after writing the actual time, what is the PC and Status register contents?

What code was running while you stop execution the handler, the main process or the void process?