Lab 14: System calls (I)

Introduction and goals

The MIPS architecture contains the elements that enable implementing a general-purpose operating system: operation modes, exceptions, coprocessor 0, etc. In labs 14 and 15, you will complete a rudimentary operating system called MiMoS (*MIPS Monitor System*). The starting point for MiMOS is like the handler you developed in lab 13. That handler only handled interrupts INT0* and INT2* (keyboard and clock). In this session, you will also implement support for system functions and a rudimentary form of concurrency.

In particular, the goals are:

- Implement new system functions accessible to user programs via the *syscall* instruction. An initial handler version is provided, *MIMOSv0.handler*. This version 0 of MiMoS already implements two system functions: *get_version* and a polling implementation of *print_char*.
- To implement a simple process management mechanism. Some system calls require suspension of the user process while some peripheral performs a needed operation. The user process will remain suspended until an interrupt request arrives from the peripheral, signalling the completion of the I/O operation. For simplicity, we will consider only one user process and the *idle* process, a system process that does nothing but consume CPU cycles while the user process is suspended.

Materials

- PCSPim-ES (the same simulator used in labs 12 and 13).
- Source code of MiMoS version 0, in file *MIMOSv0.handler*.
- Test user programs: *User0.s*, *User1.s* and *User2.s*. These files will not be modified but executed as user programs to test successive handler versions. *User0.s* uses the system calls already implemented on *MIMOSv0.handler*. *User1.s* will test *MIVOSv1.handler*, and so on.
- Three appendices: a list of MIPS system functions, details of the relevant registers in coprocessor 0, and a description of the keyboard, console, and clock adapters.

Preparation

The simulator settings must be configured according to Figure 1. Note the tick in the "Syscall Exception" option. This enables our implementation of system functions instead of letting PCSpim-ES simulate them.

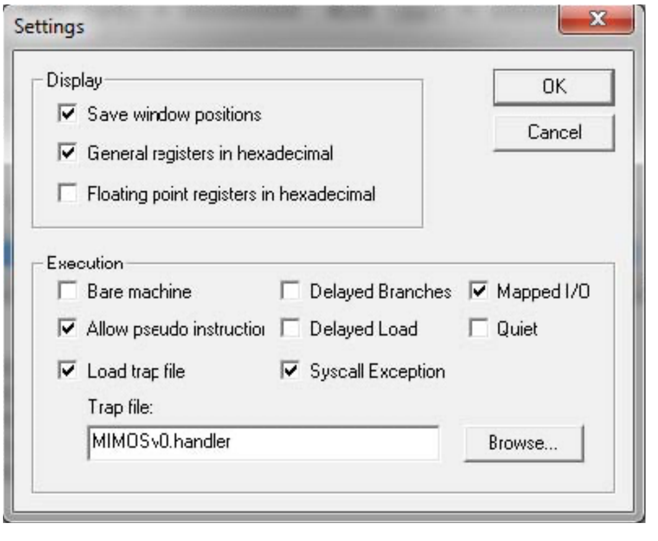

Figure 1: PCSpim settings to enable MIMOSv0.handler.

From the previous lab session, remember that you will be using two files:

- The handler or *Trap file*, with extension *.handler*, defines exception handler segments *.kdata* and *.ktext* and a *.text* fragment containing the initialisation code, a jump to the user program and a final call to the PCSpim system function *exit*.
- The user program, with extension *.s,* contains the remaining content of segments *.data* and *.text*.

Every time a user program is opened (*File>Open*) or reloaded (*Simulator>Reload*), the simulator will also load the file indicated as *Trap File* in the settings window (see Figure 1).

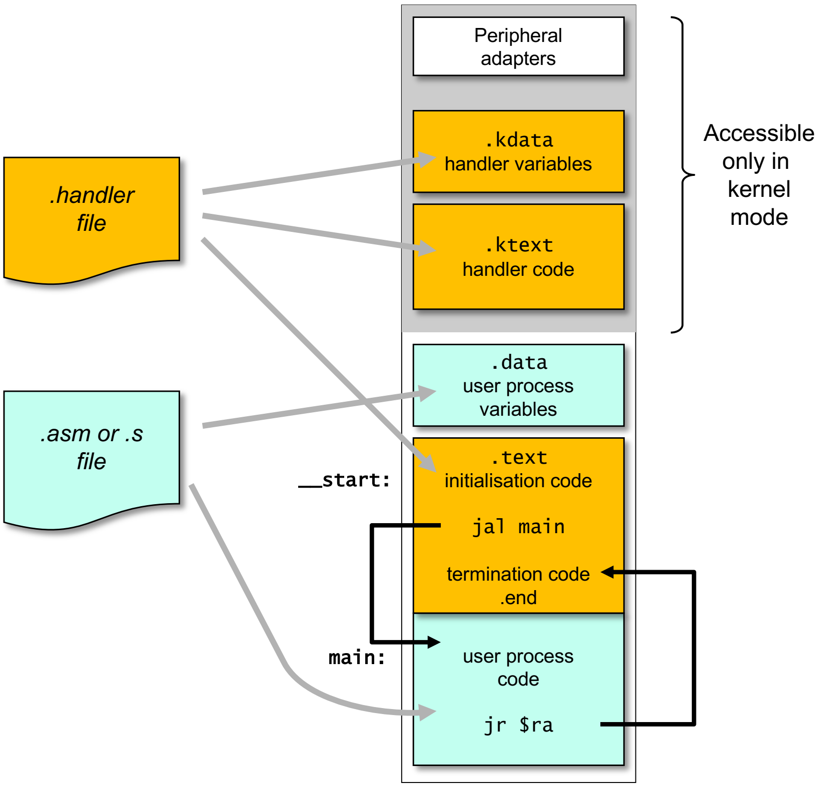

Figure 2: Correspondence between the handler and user files and the system's memory layout. (Reminder from previous lab session)

System initialisation involves three steps:

1. Prepare the peripherals, i.e., enable or disable interrupt requests in their respective adapters.
2. Configure the exception coprocessor `$Status` register: interrupt masks, execution mode and global enable/disable of interrupts.
3. Transfer control to the user program.

Step 3 is implemented by `jal main`, so the user program must have an entry point marked by the label `main`. The user program ends its execution with `jr $ra` (see Figure 2).

Structure of MiMoS handler

Figure 3 describes the handler structure. The blocks in this figure appear in the same order as in the source file *MiMoSv0.handler*. The handler structure is like the result of the previous lab session, but it also includes the handlers of two system functions, *print_char* and *get_version*. You will add more functionality to this exception handler in this lab session.

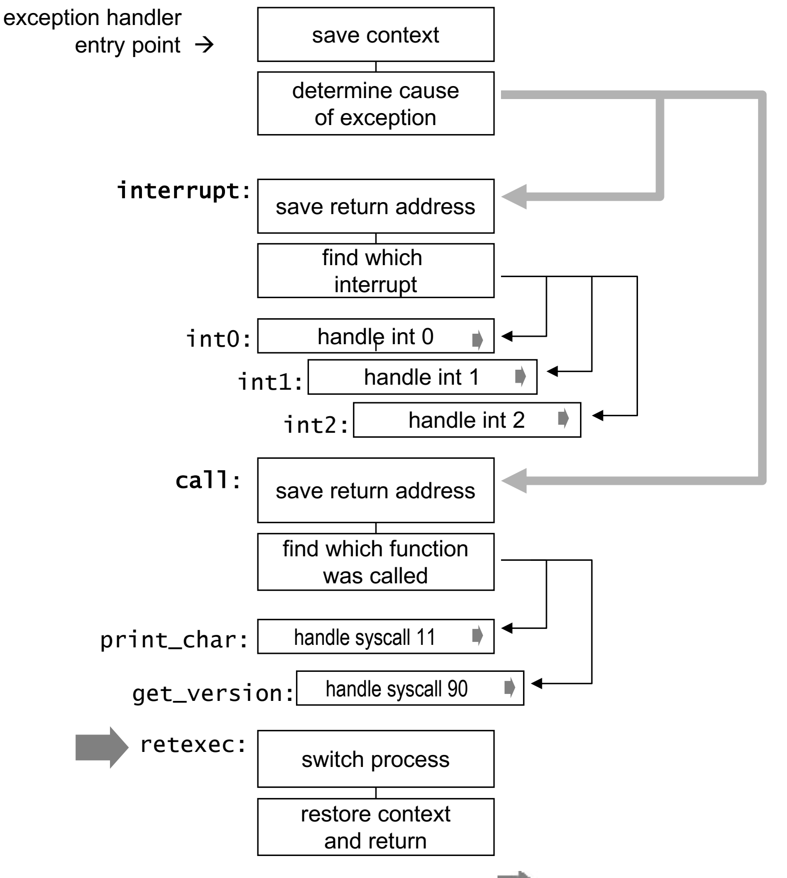

Figure 3: Structure of the MiMoS handler. The symbol ➡ indicates a branch to retexc.

After saving the context, the handler identifies the cause of the exception (among peripheral interrupts or system calls). In both cases, after saving the return address, a section of identification (interrupt line or function) jumps to the label where the corresponding handler starts. Every handling section must end with "b retexc" to jump to the final exception handler section to (potentially) switch the running process, restore the context, and return.

In the context switch section, whether to return to the user process (if it is READY) or to the idle process (if it is WAITING) is decided. The return address is found in the kernel variable retaddr in the first case. In the second, the handler returns to the idle process (a single branch instruction to itself).

In *MiMoSv0.handler*, all interrupt handling blocks remain to be implemented. In the initialisation section, interrupts are globally enabled, but all interrupt lines are masked and disabled in their corresponding adapters. Some system functions must be added in successive versions as well. Only the function get_version and a polling version of print_char have already been implemented in MiMoS version 0. There are labels defined throughout the handler for inserting the missing parts.

Task 1. MiMoS v.0: System functions *get_version* and *print_char*

MiMoSv0.handler only implements two system calls: 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 to print	Character printed on console

Table 1. System calls already implemented in *MiMoS v.0*. The function print_char uses polling.

- Inspect the code of MiMoS v.0 and check that it conforms to the structure of Figure 3. Find the implementation of the two services. Note that get_version returns $v0 = 0 and that print_char writes a character to the console using polling synchronisation. During this lab session, you will develop new versions of MiMoS, and for each version, you will need to modify the value returned by get_version consistently to make it return the version number you are working on.
- Find the system initialisation code. Since *MiMoS v.0* doesn't handle interrupts, the clock, keyboard, and console interrupts are disabled in their adapters (described in Appendix 3) and masked in the exception coprocessor `$Status` register (Appendix 2).
- To test *MiMoSv0.handler*, use the program in *User0.s*. This program calls the functions of Table 1 to obtain and print the system version, and then it enters an endless loop to print successive integer values. *User0* fills the console with the following text:

```
MiMoS v.0
1
2
…
```

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

Question 1. Stop the simulation while running *User0.s*. Before clicking the button to stop the program, pay attention to the simulator message "*Execution paused by the user at PC* = ...". If the PC points to a user program instruction (an address of type "0x0040*nnnn*"), then press "Yes" to enforce the break. If it points to a kernel instruction (an address of type "0x8000*nnnn*"), then press "No" and try again. We want to break while the user process Main is running.

- What is the value of `$Status` (coprocessor 0 register $12)? Check it on the upper simulator panel.

  3

- Is the processor in kernel or user mode? Are interrupts globally enabled?

  The 2 least significant bits are enabled, so interrupts are enabled (IEc = 1), and the processor is in user mode (KUc = 1).

- What is the value of the interrupt mask bits in the Status register?

  All are 0

- What instructions within the initialisation part of *MiMoSv0* disable the keyboard, clock, and console interrupts?

  li $t0, 0xffff0000
  sb $zero, 0($t0) # Disable KEYBOARD hardware interrupt

  li $t0, 0xffff0008
  sb $zero, 0($t0) # Disable CONSOLE hardware interrupt

  li $t0, 0xffff0010
  sb $zero, 0($t0) # Disable CLOCK hardware interrupt

- Which instructions initialise the `$Status` register?

mfc0 $t0, $12
ori $t0, $t0, 0x0003 # Enable ints., but keep all masked. Set User mode
mtc0 $t0, $12

From now on, you will develop successive versions of MiMoS incorporating new system functions. **Create every new version from the previous one, making a copy of the handler code file and renaming it with the latest version number**.

Task 2. MiMoS v.1: System function *get_time*

In MiMoS v.1, you will add the system function get_time, which returns the current system time in seconds, counting from the start of the execution. The function is specified in Table 2, along with the already implemented services of version 0.

Function	Code	Arguments	Results
get_version	*$v0* = 90		*$v0* = version number
print_char	*$v0* = 11	*$a0* = character to print	Character printed on console
get_time	*$v0 = 91*		*$v0* = current time, in seconds

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

- Save the file *MiMoSv0.handler* with the name *MiMoSv1.handler* and work on this new file. Modify the implementation of `get_version` to make it return "1" as the system version.
- Within the *kdata* section, in the "*Clock variables*" part, add a new variable named *seconds* of type *word* and give it an initial value of zero. This variable will count elapsed seconds.
- In the interrupt handling part, from label `int2`, write the clock interrupt handling code that will be executed every second. It must increment the variable *seconds* to account for one more second elapsed and cancel the clock interrupt in the clock adapter (described in Appendix 3). In pseudocode:

```
int2:    seconds = seconds + 1
         cancel interrupt
         b retexc
```

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

int2:

```
lw $t0, seconds
addi $t0, $t0, 1
sw $t0, seconds

li $t0, 0xFFFF0010
li $t1, 1  # Cancel interrupt, keep interrupts enabled
sb $t1, 0($t0)

b retexc # end
```

Question 3. Modify the initialisation code to allow clock interrupts. You must enable them in the clock adapter and unmask them in the $Status register of coprocessor 0.

```
## Prepare coprocessor Status register and user mode

mfc0 $t0, $12
ori $t0, $t0, 0x0403  # Enable ints., and unmask clock. Set User mode
mtc0 $t0, $12
```

```
li $t0, 0xffff0010
li $t1, 1
sb $t1, 0($t0) # Enable CLOCK hardware interrupt
```

- Now that the clock interrupt handler keeps the seconds variable updated, you are ready to complete the implementation of the system function `get_time`. The handling is like `get_version`, but you must load *$v0* with the value of the variable *seconds* instead of the (constant) version number.

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

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

```
get_time:
    lw $v0, seconds
    b retexc
```

- Time to run and check version 1. Modify the simulator settings to use *MiMoSv1.handler* as the *Trap File* and load *User1.s* as the user program. *User1* does the following:

  Print MiMoS version number
  Repeat forever
      compute something
      call `get_time`
      print current time

The "compute something" part is an empty loop that consumes CPU time between calls to get_time.

Question 5. Can *User0* execute correctly with handler *MiMoSv.1*? Can *User1* execute correctly with handler *MiMoSv.0*? Explain your answers.

MiMoS processes

We can abstract away the program contained in the user file and name it the *main process*, or *Main*. So far, *Main* has always been active, but you must consider other states hereafter. The state of *Main* is stored in the handler variable state. There are two states defined using two respective constants:

```
## Possible states of Main
        READY = 0
        WAITING = 1
state: .word READY        # State of Main, initially READY
```

An *idle process* is implemented in the text section of the handler file. This is a helper for implementing context switching: whenever *Main* enters the WAITING state, MiMoS will switch to this process. The void process code is the following:

```
idle_process: b idle_process
```

Being a single instruction that jumps to itself, this process is always ready. Moreover, it does not use any registers. Hence, its context is just its starting address, a constant the handler knows as label idle_process.

Every time the MiMoS handler is invoked (because of an exception), it needs to know where to return after handling that exception. At the end of the exception handler (label `retexc`) is the process handling code that sets $k0 to the proper return address, hence resuming one process or the other: *Main* is resumed if it is READY, or the idle process is taken otherwise:

if (state == READY)
    $k0 = main process returning address
else
    $k0 = idle_process
end if

Note that a context switch is not required in MiMoS. This is because the idle process does not have a context to be preserved. Hence, you only need to maintain the context of the main process. During the handler execution, the PC is stored in *retaddr,* and the three general-purpose registers `$at`, `$t0`, and `$t1` are saved to *savereg* so the handler can use them. Should you need more registers for the handler, you can add more space for them in the context area (after label *savereg*).

Task 3. MiMoS v.2: System function *wait_time*

In version 2 of MiMoS, you will add the system function `wait_time` described in Table 3. This system call allows the main process to suspend itself during a specified time interval. It is like the UNIX system function sleep().

Function	Code	Arguments	Results
wait_time	*$v0 = 92*	$a0 = time in seconds	Caller suspended for *$a0* seconds

Table 3: New blocking function to be included in MiMoS v.2

When *Main* calls wait_time, it gets suspended during the time given as an argument in $a0. Therefore, the code for this function should set the state of *Main* to WAITING and allocate the CPU to the idle process (since there are no more processes in MiMoS). Consequently, after executing this function, the exception handler will not return to the main process but to the idle process. The clock interrupt handler will restore *Main* to the READY state when the time specified in $a0 has elapsed since the call to wait_time. You must declare the kernel variable alarm to store the time when *Main* must be resumed. Do it in the area reserved for clock variables in *kdata*.

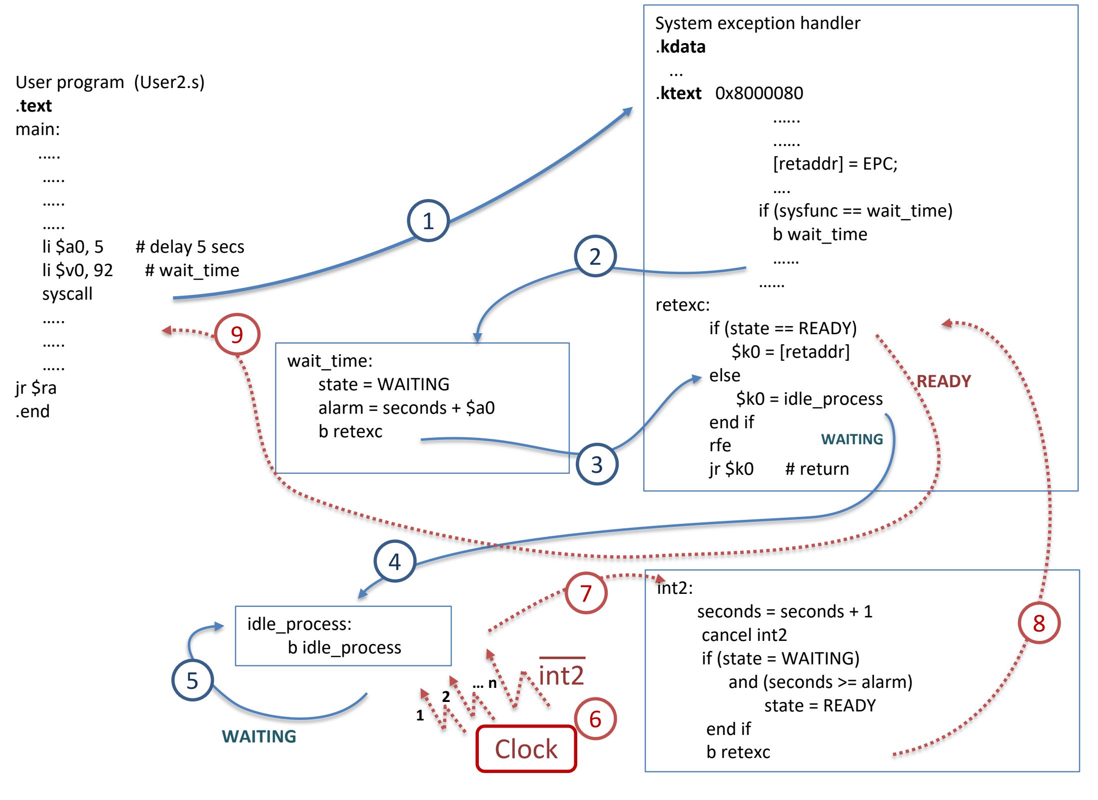

Figure 4. Process switch when User2 calls *wait_time*.

The events represented in Figure 4 are explained in more detail here:

1. *Main* calls the system function 92 (*wait_time*) with an argument of 5 seconds.
2. The exception handler stores the return address of the main process in the variable *retaddr,* checks that the called function is 92, and then it jumps to handle *wait_time*.
3. *wait_time* changes the state of the main process to WAITING and then jumps to *retexc* to return.
4. The returning code finds that the main process is not READY and returns to the idle process.
5. The idle process runs, doing nothing but jumping to its only instruction.

6. Several clock interrupts occur. On every interrupt, the variable seconds is incremented by 1.
7. When "n" interrupts happen ("n" is five in Figure 4), the suspension time has expired, and the handler changes the state of *Main* to READY and jumps to the *retexc* section to return.
8. The returning code notices that the main process is READY, so the return address is taken from *retaddr.*
9. The CPU is allocated back to the main process.

- Make a copy of the *MiMoSv1.handler* file and save it as *MiMoSv2.handler*. In this new file, modify the implementation of `get_version` to make it return the value 2.
- The function `wait_time` must set the state of the main process as WAITING and compute and store an alarm time when *Main* should be resumed. The alarm time must be stored in a new kernel variable alarm, and it can be easily calculated as the current time (seconds) plus the delay specified by the caller in $a0.

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

Question 6. Write your implementation of *wait_time*.

```
wait_time:
    # Set state to waiting
    li $t0, WAITING
    sw $t0, state

    # Configure alarm
    lw $t0, seconds
    add $t0, $t0, $a0
    sw $t0, alarm

    b retexc
```

► To complete the required functionality, the clock interrupt handler (starting at label `int2`) must increment the variable seconds by one unit, as before. In addition, if the main process state is WAITING and the value of *seconds* equals *alarm*, it must change the main process state to READY.

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

Question 7. Write the clock interrupt handling code.

```
int2:
    # Update seconds
    lw $t1, seconds
    addi $t1, $t1, 1
    sw $t1, seconds

    # Cancel interrupt (but keep clock interrupts enabled)
    la $t0, 0xffff0010
    li $t1, 1
    sb $t1, 0($t0)

    lw $t0, state
    li $t1, READY
    # If the state is READY, return
    beq $t0, $t1, end_int2
    # At this point in the code, the state is WAITING
    # If seconds < alarm, return
    lw $t0, seconds
    lw $t1, alarm
    blt $t0, $t1, end_int2
    # At this point in code, state = WAITING && seconds >= alarm
    # Switch the state to ready
    li $t0, READY
    sw $t0, state
```

- To check version 2 of MiMoS, modify the simulator settings to use MiMoSv2.handler as the Trap File and load *User2.s*. This is a test program that does the following:

  Print MiMoS version number
  Repeat forever
  - *call* `get_time`
  - *write current time*
  - *call* `wait_time` *(5 seconds)*

Question 8. Stop *User2.s* just after it writes the current time to the console (*n* seconds). What are the contents of the PC and $Status registers?

PC = 00400040

Status = 00000403

What process did you stop, *Main,* idle_process, or the system exception handler?

idle_process