Lab 15: System calls (II)

This session is a continuation of the previous one. Hence, they share the same goals. The starting point for this session is the handler available in *MiMoS_v3.handler*. This file contains the implementation of four system functions: *get_version*, a polling implementation of *print_char, get_time* and *wait_time*. You will implement a new system function, *read_char,* and modify *print_char* to make it use interrupt synchronisation with the console instead of polling.

Materials

- PCSpim-ES.
- The file *MIMOSv3.handler* will be your starting point. You will extend this file to implement the functions we want in MiMoS version 3.
- The file *User.s* contains the code of the user process that you will not need to modify.

Processes in MiMoS

Recall from the last session that MiMoS uses the so-called *idle process* to facilitate context switch:

```
idle_process:   b idle_process
```

This is the process to switch to when the user process is waiting, i.e., when it calls a suspending system function such as *wait_time* from Lab 14. The idle process is always active and has no context to preserve. The program counter constantly points to the label idle_process when the idle process is running.

The process switch is decided in the final part of the exception handler (label retexc). The proper return address is saved to register $k0, and the handler returns with jr $k0. There are two possibilities: if the user process is READY, it is resumed; if it is waiting due to an ongoing blocking system call, the handler returns to the idle process.

```
If (state == READY)
   $k0 = return address to user process (previously obtained from $EPC)
else
   $k0 = address of idle_process
end if
```

Remember that, in MiMoS, **context switches** are straightforward: you only need to preserve the context of the main process because the idle process does not use any registers.

Task 1. Adding the system function read_char

In the file *MiMoSv3.handler,* you will find the implementation of system functions *get_version*, *get_time*, a polling version of *print_char*, and *wait_time.* To this set of functions, which was the result of Lab 14, you will now add the system function *read_char, which* reads a character from the keyboard. In Task 2, you will implement a new version of *print_char* using interrupt synchronisation.

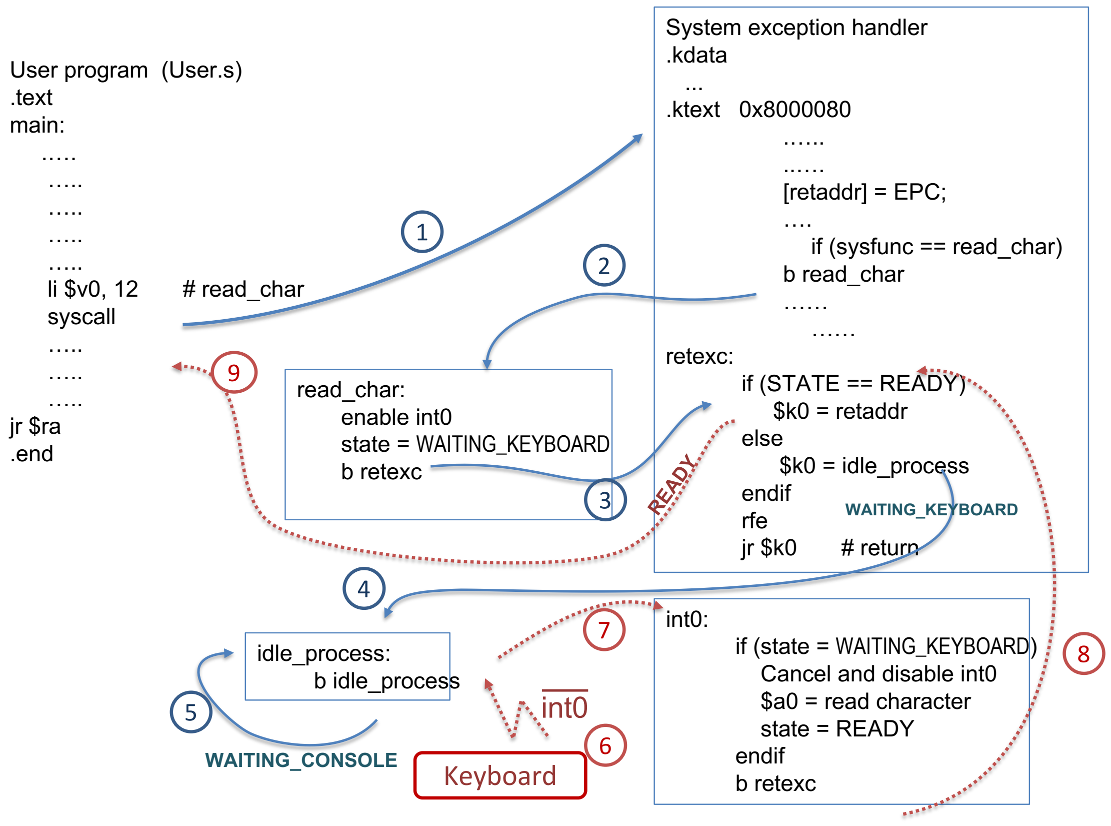
Function	Code ($v0)	Arguments	Results
get_version	90		$v0 = Version number
get_time	91		$v0 = Current time in seconds
wait_time	92	$a0 = a time in seconds	Suspend until *now*+$a0 secs
read_char	12		$a0 = character read
print_char	11	$a0 = character to print	

Table 1: The five system functions of MiMoS v.3. The first three are already implemented. Task 1 is about implementing read_char, and Task 2 is about reimplementing the polling version of print_char to use interrupt synchronisation with the keyboard.

The function *read_char* is blocking, i.e., it must suspend the caller until there is a keyboard interrupt (effect of a key press on the keyboard). The user process must be resumed at that time, and it will find the typed character in register $a0. The keyboard interrupt handler must set the state of the user process to READY so that it is resumed after a keypress is detected.

Keyboard interrupts must be enabled only during an ongoing call to *read_char*. To simplify the enabling/disabling keyboard interrupts, the interrupt request line *int0** will remain permanently unmasked on the coprocessor Status register. Hence, keyboard interrupts will be enabled/disabled via the Interrupt Enable bit of the keyboard adapter only.

The sequence of events that occur while waiting for the return from the function *read_char* is shown in Figure 1. As in the previous lab, the idea is to change the caller's state from READY to WAITING_KEYBOARD. After this change, the handler does not return to the user process but to the idle process instead. The keyboard interrupt handler will restore the user process state to READY, which in turn will cause the return of the exception handler to the main process.


```
User program  (User.s)
.text
main:
    .....
     .....
     .....
     .....
     .....
     li $v0, 12        # read_char
     syscall
     .....
     .....
     .....
jr $ra
.end
```

```
System exception handler
.kdata
  ...
.ktext   0x8000080
              ......
              ......
              [retaddr] = EPC;
              ....
                  if (sysfunc == read_char)
              b read_char
              ......
                  ......
retexc:
      if (STATE == READY)
          $k0 = retaddr
      else
           $k0 = idle_process
      endif
      rfe
      jr $k0       # return
```

```
read_char:
      enable int0
      state = WAITING_KEYBOARD
      b retexc
```

```
idle_process:
      b idle_process
```

```
int0:
      if (state = WAITING_KEYBOARD)
         Cancel and disable int0
         $a0 = read character
         state = READY
      endif
      b retexc
```

Figure 1: Execution flow upon a call to *read_char*.

► Taking the file *MiMoSv3.handler* as a starting point, implement the handler for the system function *read_char*. The handler must put the user process in the state `WAITING_KEYBOARD` and enable the keyboard interrupt in the keyboard adapter. Note that a numerical value is assigned to the constant `WAITING_KEYBOARD` in *MiMoSv3.handler* so that it differs from other possible blocking causes (see towards the beginning of the *kdata* section).

```
read_char:    Enable keyboard interrupt (in the keyboard adapter)
              state = WAITING_KEYBOARD
              b retexc
```

Question 1. Copy here your implementation of *read_char*.

```
read_char:
      # Enable keyboard interrupt
      li $t0, 0xFFFF0000
      li $t1, 2
      sw $t1, 0($t0)

      # Set state to waiting keyboard
      li $t0, WAITING_KEYBOARD
      sw $t0, state

      b retexc
```

► To complete the functionality of *read_char*, write the keyboard interrupt handler (after the label `int0`). The handler must check that the user is in the state `WAITING_KEYBOARD`. If true, it must read the character from the keyboard into *$a0*, which implicitly cancels the keyboard interrupt. Then, it must disable the keyboard interrupt and set the user process state to READY.

Note that this function reads the keyboard but does not echo the typed character to the console.

```
int0:  If (state == WAITING_KEYBOARD)
           $a0 = character from the keyboard
           cancel and disable the keyboard interrupt
           state = READY
       end if
       b retexc
```

Question 2. Copy your implementation of the keyboard interrupt handler here.

```
int0:
     # Only continue if state == WAITING_KEYBOARD
     lw $t0, state
     bne $t0, WAITING_KEYBOARD, int0_end

     # Cancel and disable keyboard interrupts
     li $t0, 0xFFFF0000
     sb $zero, 0($t0)

     # Get character from keyboard and store it in $a0
     lb $a0, 4($t0)
     ...

     ...
     # Set state to ready
     li $t0, READY
     sw $t0, state

     int0_end:
     b retexc # end
```

► In the system initialisation code, you must enable interrupts and unmask the interrupt lines *int0** for the keyboard, and *int2** for the clock. Write the mask value in hexadecimal (bits 8 to 15 of register `$Status` in coprocessor 0).

```
li $t0, 0x0503      # Interrupts globally enabled, int0 and int2 unmasked
mtc0 $t0, $12
```

► Test the handler using the program *User.s*. The main program in this file does the following steps:

```
Display OS name and version
Repeat forever:
    get_time
    Print the current time to the console
    read_char
    Print the read character to the console
```

Task 2. Modifying the system function print_char

To complete *MiMoS* v.3, you will now modify the implementation of *print_char* (see Table 2) to make it use interrupt synchronisation instead of polling. The scheme to apply is the same as in *read_char*: the console interrupt must only be enabled from when the program calls ***print_char*** until the peripheral (the console) becomes ready. When *Main* calls print_char, the handler must set the state to WAITING_CONSOLE. The console interrupt handler will print the character to the console and set the user process state back to READY.

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

Table 2: Specification of the system function print_char. (Last row of Table 1)

► Implement the code of *print_char*. The new handler must put the user process in the state WAITING_CONSOLE and enable interrupts in the console adapter.

```
print_char:
    enable console interrupts
    State = WAITING_CONSOLE
    b retexc
```

Question 3

Write the code for the system function *print_char.*

```
print_char:
    # Enable console interrupts
    li $t0, 0xFFFF0008
    li $t1, 2
    sb $t1, 0($t0)

    # Copy character to print (in $a0) to the data register of the console
    sw $a0, 4($t0)

    # Set state to waiting console
    li $t0, WAITING_CONSOLE
    sw $t0, state

    b retexc
```

► You must write the console interrupt (label `int1`) handler code, which should only apply if the process is found in the state WAITING_CONSOLE. Since an interrupt *int1* signals the readiness of the console, the handler must copy $a0 to the console's Data register, cancel and disable console interrupts, and set the user process to the READY state.

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

Question 4

Copy here the code corresponding to the console interrupt handling.

```
int1:
    # Only continue if the state is WAITING_CONSOLE
    lw $t0, state
    bne $t0, WAITING_CONSOLE, int1_end

    # Cancel and disable console interrupts
    li $t0, 0xffff0008
    li $t1, 0
    sb $t1, 0($t0) # disable the interrupts from the keyboard

    li $t1, READY
    sw $t1, state

    int1_end:
    b retexc # end
```

► All that remains is to prepare the interrupt mask in the handler initialisation section so that console interrupts remain unmasked. Any previous user program can be used to test the new handling because they all use the function `print_char`.

Type the mask value in hexadecimal.

```
li $t0, 0x0703        # Interrupts globally enabled, int0 and int2 unmasked
mtc0 $t0, $12
```

FINAL REMARKS

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

- PCSpim does not fully simulate the processor kernel mode, so the user program can, for example, execute privileged instructions and direct access to the peripheral interfaces without calling system code (MiMoS).
- PCSpim does not simulate virtual memory. This is the reason why MiMoS cannot implement memory protection mechanisms.
- MiMoS is written in assembly, so adding features typical of actual operating systems that require careful programming and complex data structures would be hard.

Relevant operating systems have been written with similar restrictions. CP-M and DOS were operating systems that used a similar scheme to MiMoS. Indeed, the implementation of those OS was carried out by a team using development tools designed for the job.