Multitasking system with priorities

Multitasking systems, and in particular those intended for use in real-time applications, often use a priority mechanism, where all tasks are assigned a priority value. The scheduling principle is simple and consists of defining that a lower priority process will only execute if there is no higher priority process available to occupy the processor. For the sake of simplified management, we will only consider two priority levels, high (1) and low (0). We will maintain a 'execution' list that will only have the process in this state, but we will have two 'readyX' lists, one for each priority X.

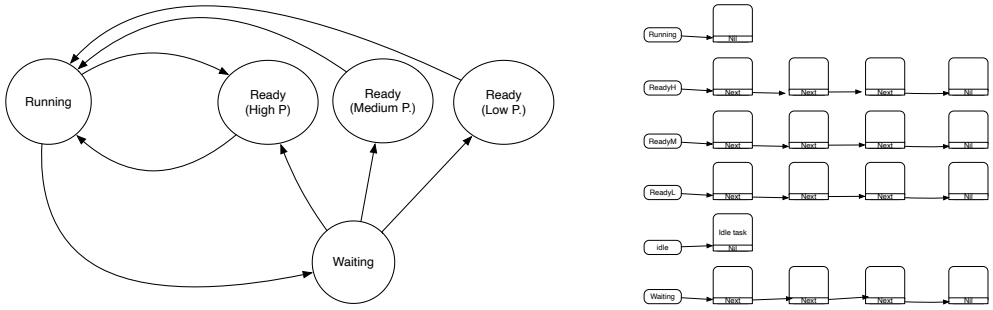

Figure B.1: Left: Status of the processes and possible transitions. Right: Processes queues.

For lower priority processes to have the opportunity to function, and what is called 'starving' not to occur, it is necessary that they self-suspend from time to time. This suspension is done at the expense of a system call that either serves to suspend the process (sleep) or is a request for an operation that can take several ticks of the clock to be completed. The tasks return to the 'readyX' state, in the first case when the requested suspension

period ends, and in the second case when the request has been completed (e.g. data received from a peripheral).

When a process is suspended, it transitions from the 'running' to 'waiting' states as shown in figure K.1, where you can also see the different process queues. Notice that the 'idle task' contains just a loop cycle, eventually updating some counter, and is part of the system, therefore not not added by the programmer. Can you think of a reason why it is necessary to have it?

B.1 Protection levels and operating system calls

Operating systems provide diverse services such as file access, memory management and process management, among others. Access to these services is done through the so-called 'system calls', which, from a programmer's point of view, correspond only to the call of a function of the language in use, but which makes use of a special instruction to activate the requested service. This instruction produces an interrupt (or exception) by immediately jumping from the user process code to the interrupt service code.

An example of the services provided by the operating system is the creation of processes that, contrary to what was done in the previous assignment, is not normally done by using functions at the user side (User Level) but rather at the operating system level (Kernel level).

What we use is typically some library of functions that on the one hand provide a user-friendly programming interface and on the other hand, prepare the input parameters for the system call mechanism of the processor in use.

In the case of MIPS, we have the syscall instruction in which its use generates an exception, that is, it calls the service routine for exceptions and interruptions, which starts by analyzing what happened to have it activated and depending on the value in the defined register to choose the service and other parameters to perform the respective functions.

Unfortunately, MARS does not allow SYSCALL to be handled by adding code written by us as would be desirable, making it impossible to increase the set of pre-defined system calls. To overcome this limitation we will use another mechanism that will allow similar operation.

So instead of the syscall instruction, we will use one of the instructions that allows us to generate exceptions. The instruction to use will be

```
teqi $zero, 0
```

and when this is executed the interrupt and exception service routine will be called, but (according to MARS help) bits 2-6 of the cause register will contain the value 13. So we can create any service using the registers $v0 and $a0 and $a1 as with the syscall instruction and associated services.

B.2 Work to be done

Exercise B.1
Add a field to the PCB called TICKS_TO_SWITCH that will be initialized to the value 3 when the task is created. When an interruption occurs, this value must be decremented by one unit and when it reaches zero, tasks must be switched. When each task transitions from ready to running state this value is reset to 3. 10%

Exercise B.2
Replace the keyboard interrupt with the periodic interrupt generated by Digital Lab Sim. Since these interrupts can occur very frequently, we will generate a TICK for every 4 interrupts received from this source. To do this, read its help. 10%

Exercise B.3
Change the `prep_multi()` function so that, following exactly the order in the first work, it now starts the necessary structures to accommodate the two priority queues. 10%

Exercise B.4
Change the code of the interrupt service routine so that you can differentiate the interrupts generated by the inclusion in the code (main) of the 'teqi $zero, 0' instruction, from other interrupts such as those generated by peripherals such as the keyboard used in the previous work. 10%

Exercise B.5
We will now use a new function to create tasks, which in C would be equivalent to

```
void newtask(void * entry, int priority);
```

Noting that the stack must be started correctly, the code for this function is as follows:

```
newtask:
    added $sp, $sp, -4
    sw $a2, 0($sp)
```

```
    move $a2, $a1
    move $a1, $a0
    li $a0, 1
    teqi $zero, 0
    lw $a2, 0($sp)
    added $sp, $sp, 4
    jr $ra
```

To do this, add the interrupt service routine in the aforementioned case, the necessary code so that when the value in $a0 is 1, create a new task whose starting address is received in register $a1 and the priority in register $a2.

Note that this consists of adapting the task creation code from the previous work to this new method.

20%

Exercise B.6

Likewise, we will use the routine below, which is equivalent to the following function in the C language

```
void sleep(int numtics):
```

```
sleep:
    addiu $sp, $sp, -4
    sw $a1, 0($sp)
    move $a1, $a0
    # the sleep time is passed in $a1
    li $a0, 2
    teqi $zero, 0
    lw $a1, 0($sp)
    addiu $sp, $sp, 4
    jr $ra
```

- To do this, modify the interrupt service routine with regard to the response to the exception caused by the 'teqi' instruction, so that when the value in $a0 is 2, it suspends the execution of the task that called it for the number of tics specified through the $a1 register.
- When this case occurs, you must change the task in 'running' to the 'wait' state, and store the number of waiting tics in an additional field on the PCB.
- Then select the task from the highest priority list to be in the 'running' state. If there is no task in the 'readyX' lists, a special task 'idle' must be started.

20%

This idle task must also have a PCB created when the `prep_multi()` function is called and the pointer to it will be stored in a variable "idle_p". To distinguish "normal" tasks from idle ones, we can add a field to the PCB,

which is the task ID, where the task with ID=0 is idle, the "main" task will have ID=1 and the rest have IDs whose value will always be increasing as they are added.

Note that the idle task can never move to the ready list.

Exercise B.7

Modify the interrupt service routine so that for each 'TICK' the behaviour becomes the following:

- for each task in the list 'waiting' decrease the amount of time that this task must still be suspended by one unit. If this value is zero, the task must return to the end of the list corresponding to its priority.
- If there are in the lists 'readyX' a priority higher than or equal to that of the task in 'running' state, then the tasks must be exchanged by executing the first task in the queue with the highest priority and moving the one which goes to the end of the respective priority list.
- remember that if the task that exits is the idle task then it has its own list and should not be added to any other.

20%

Exercise B.8

Write an example test in file *main1.s* in which there is only the main task that alternates between cycle periods of 100 repetitions (printing a message on the screen) and sleep cycles of 3 ticks.

Mandatory or -50%.

Exercise B.9

Write another example test in file *main2.s* with at least 3 tasks , including:

Mandatory or -50%.

- a task, written by you, that does something of your choice, interleaved with sleep calls of different durations.
- a task, written by you, that continuously fills the *bitmap display* with varying colors, interleaved with sleep calls.
- a task for which the code is (will be) provided in a file named *mystery.s* .

B.3 Bonus

If you complete the previous paragraphs, and only in that case, you can access a bonus that can compensate for losses in points in the previous assignment if carried out and delivered.

Exercise B.10

In operating systems, it is normal to have the possibility of consulting the "processor load". This load corresponds to the percentage of time that the processor is executing tasks or, in other words, is not executing the idle task. Let's use a digit from "digital lab sim" to indicate the processor load. To do this, we will count in each period of 18 interruptions how many of them the idle task was being executed. At the end of the 18 interruptions, we subtract this value from 18 and divide it by 2. The result should be shown in the left digit of the digital lab sim.

10%

Exercise B.11

Write a task that prints the an increasing counter value, and its factorial calculated recursively. As the factorial value quickly "explodes", reset the counter to 0 when it reaches 20. This task should have low priority and pause for 5 ticks each time it prints the calculated value.

10%

B.4 Delivery

Deliver the properly commented code and a report of up to 4 pages in PDF or Word format. Both the code and the report must identify the authors. Delivery must be made through a ZIP file which, when decompressed, must allow immediate execution by MARS.

B.5 Defense and Plagiarism

The work will be subject to a mandatory oral presentation by all members of the group and everyone must be able to explain the operation in detail.

Particular attention will be paid to plagiarism. Works that have the same structure will have the scores divided by the number of identical or very similar works. This comparison will be made line by line. Therefore, the answer to each item must be clearly identified, otherwise, they will not be considered.