Interrupt-Based Multitasking

A

Nowadays, all operating systems exploit parallel or pseudo-parallel execution of tasks, taking advantage of a long-identified fact: except for rare pure computational applications, most applications make very limited use of the processor, spending most of their time waiting for data to become available, for produced data to be consumed, or simply for time to pass to ensure a certain operating cadence. For example, an application that simulates an analog clock does not need to execute the pointer drawing update cycle at the maximum speed, as the pointers only move from second to second.

Therefore, we can share the processing capacity of a processor among a set of tasks, allowing all of them to make progress in their execution.

A.1 Process States

By what has been said above, the change in the execution state of tasks (or processes) can be described using the scheme in figure Fig.A.1.

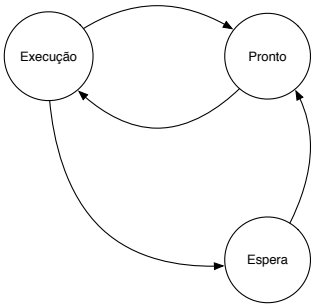

Figure A.1: Process states and possible transitions.

Transitions between the **Execution** and **Ready** states usually result from the expiration of the maximum time a process can be in the first state without voluntarily suspending itself. Here, when one process 'exits', another 'enters', in a cyclic swapping mechanism often called *round-robin*.

The case in which a process enters the **Wait** state is a consequence of a request by the system to perform a time-consuming operation (in terms of CPU time), such as writing to a file, communicating with a peripheral, or explicitly requesting a pause of a few seconds.

Typically, these state transitions are invisible to the processes; in other words, processes execute as if they were always in the same state, meaning there is nothing to be done from the perspective of the common programmer to handle these state transitions. Let's now examine what the states correspond to, and what can trigger the transition between them.

Execution This state is assigned to a process (or several processes, in the case of a multi-processor system) that is running on the CPU. The instructions of this process are being executed as if there were no other processes. The transition from this state to the **Wait** state can occur due to a request to the operating system to suspend it for a certain period or to access a resource that is not yet available. The transition to the Ready state occurs when the scheduler decides that another process should start executing, based on the scheduling policies in use, such as reaching the maximum time slice or another higher-priority process becoming ready for execution after a pause.

Ready In this state, we can find all processes that are waiting to be placed into execution on the processor.

Wait In this state, we find all processes waiting for some event or resource. Processes that request suspension for a certain period or that wait for data or access to a resource enter this state.

A.2 Process Swapping

As mentioned earlier, swaps between the **Execution** and **Ready** states occur through the intervention of a timer that generates interrupts at a predefined rate. These (clock) interrupts activate a critical part of the operating system, the scheduler, whose responsibility is to check whether the currently executing process has exhausted its maximum execution time of its current assigned slot. In that case, it swaps this process with another that is ready to execute.

Therefore, we can have a 'clock' that generates an interrupt (tick) at some fixed rate, e.g. 50Hz that corresponds to a tick every 20ms. However, the operating system architect may define that forced process swaps should only occur every N ticks.

For each interrupt occurrence, the scheduler must decrement the number of ticks that the currently executing process is entitled to. If this value reaches zero, the process must be taken out of execution, moving it to the **Ready** state, and another process from this state will enter the **Execution** state.

As previously said, this process swap must be completely transparent to the processes. In other words, a process that has been placed in the **Ready** state and returns to the **Execution** state should continue to execute as if nothing had happened. In other words, the programmer should not have to worry about this.

A.3 What Information Should Be Saved During Process Swapping?

If we consider that swapping can occur due to an interrupt generated by a timer at any point in the code of the process in the **Execution** state, the solution can be seen as making a copy of the processor's state, which will be restored later. In other words, we need to save all processor registers that could potentially be used by a process to be later restored.

Therefore, each process should have a memory structure where copies of the registers, and possibly other useful information about the process, are stored. This structure is often called a Task Control Block (TCB) or Process Control Block (PCB) and is typically organized as linked lists. Figure A.2 illustrates what one of the Ready or Wait process lists could look like.

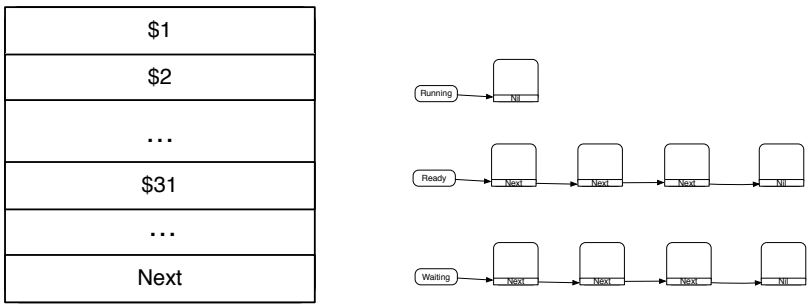

Figure A.2: Left: Example of a minimalistic PCB; Right: Process queues.

Note that the PCB for each process typically contains more information fields than the space for storing registers. This additional information may include the process's assigned name, priority, information about memory usage, the status of inputs and outputs used by the process, information about files used, and more.

A.4 Preparation

Exercise A.1
Review the concepts of interrupts. To do this, review your notes during classes and carefully read the 'Interrupts' worksheet.

Exercise A.2
Execute the code previously provided in the example of interrupt usage with a keyboard and display, and try to understand its operation. Activate the 'Keyboard and Display MMIO Simulator' tool before testing.

Hint: Use the 'Run speed at max (no interaction)' setting to step through about 2 instructions per second and run the program. Observe what happens when you press a key after the program starts and enter the 'idle_loop'. Try to understand the details.

A.5 Tasks to Accomplish

The work to be done follows the approach described in the theoretical classes, involving the construction of a preemptive scheduler triggered by an interrupt from a device. We can assume that this device is a timer that generates interrupts repeatedly at a fixed interval.

In this project, you must create a multi-programming system with task switching based on a timer expiration. Normally, this timer would be provided by a periodic clock signal, but since it is not available in MARS, we will use the keyboard interrupt to simulate a 'clock tick'.

Exercise A.3
Considering the PCB structure defined in figure A.5 that will be used to keep process information needed for management and multitasking.

10% Reserve a block of memory with space enough to store 10 PCBs, and a

10% second block of memory containing 10 task stacks, after choosing a maximum size for each stack. Use the .space directive (required) and consult

mandatory the MARS help to understand its functionality.

1	$at
2	$v0
3	$v1
4	$a0
5	$a1
6	$a2
7	$a3
8	$t0
9	$t1
10	$t2
11	$t3
12	$t4
13	$t5
14	$t6
15	$t7
16	$s0

17	$s1
18	$s2
19	$s3
20	$s4
21	$s5
22	$s6
23	$s7
24	$t8
25	$t9
26	$k0
27	$k1
28	$gp
29	$sp
30	$fp
31	$ra

32	hi
33	lo
34	epc
35	Process ID
36	NEXT PCB

Figure A.3: Content of the PCBs to be used

Exercise A.4

Create a **prepare_multi()** function that should initialize the data structures for the task list. It should also initialize the pointers for the necessary linked lists ('ready' and 'execution'). 20%

Exercise A.5

Define the structure of the PCB to be used and write the code to fill its fields for the initial execution of each of the tasks to be added. Note that PCBs, when used, must be organized as nodes in a linked list.

Do this by creating a **newtask(...)** function'20%' that takes the task's starting address as a parameter.

Note: Don't forget to allocate a memory area for the stack from the previously reserved space (not with any MARS system call), whose top address should be placed in the PCB's stack pointer area, just as the task's starting address should be stored in the **epc** area.

Exercise A.6

Write the code for the interrupt service routine that allows saving the state of the current task and switching it with another ready task. This code, while inspired by the provided code, **should not have any references to the keyboard or display; otherwise, no points will be awarded**. 20%

Note: Task switching implies that the entire execution state of the task must be saved so that when it is reactivated, it continues exactly from where it left off rather than starting from the beginning.

Exercise A.7

20%

Create the **start_multi()** function that, when called, activates interrupts and multiprocessing, putting all tasks into parallel execution through time sharing. Note that only at this point will processor sharing between the added tasks and the existing 'main' task begin.

Also, allocate a PCB for the main task here and fill it with the values to be placed in the registers when the first task switch occurs between the main task and the first task in the ready list.

Exercise A.8

20%

mandatory

The code below should be in a file named 'main.asm', along with the adaptation of the interrupt processing code in a file named 'interrupt.asm', and three additional files containing the example codes listed too.

```
        .data
STRING_done: .asciiz "Multitask started\n"
STRING_main: .asciiz "Task Zero\n"
        .text
#this is the entry point of the program
main:
# prepare the structures
        jal prepare

# new_task (t0)
        la $a0,t0
        li $a1, 1
        jal new_task

# new_task(t1)
        la $a0,t1
        li $a1,2
        jal new_task

# new_task(t2)
        la $a0,t2
        l1 $a2, 3
        jal new_task

# start_mt() and continue to
# the infinit loop of the main function
        jal start_mt

        la $a0, STRING_done
        li $v0, 4
        syscall

infinit:
        # Reapeatedly print a string
        la $a0, STRING_main
        li $v0, 4
```

```
        syscall
        b infinit

# the support functions
prep_multi:
        # write your code here
        jr $ra

newtask:
    # write your code here
    jr $ra

start_multi:
    # write your code here
    jr $ra

.include "t0.asm"
.include "t1.asm"
.include "t2.asm"
.include "interrupt.asm"
```

The code of each of the tasks **must** be in a separate file to allow for easy replacement during the defense.

mandatory

```
# file t0.asm
    .data
STRING_T0: .asciiz "\nFirst Task - "
    .text
t0:
    li $t0,0
repeat0:
    la $a0, STRING_T0
    li $v0, 4
    syscall
    move $a0,$t0
    li $v0, 1
    syscall
    addi $t0,$t0,1
    b repeat0
```

```
# file t1.asm
    .data
STRING_T1: .asciiz "\nSecond Task - "
    .text
t1:
    li $t0,0
repeat1:
    la $a0, STRING_T1
    li $v0, 4
    syscall
    move $a0,$t0
    li $v0, 1
    syscall
    addi $t0,$t0,1
    b repeat1
```

```
# file t2.asm
    .data
STRING_T2: .asciiz "\nThird Task - "
```

```
    .text
t2:
    li $t0,0
repeat2:
    la $a0, STRING_T2
    li $v0, 4
    syscall
    move $a0,$t0
    li $v0, 1
    syscall
    addi $t0,$t0,1
    b repeat2
```