SIMULATOR OF A MULTIPROGRAMMED COMPUTER SYSTEM

V1

Initial tasks

Download the compressed file from the Virtual Campus.

The work to be done in the following exercises will be developed on the indicated copy of the files contained in the V1 directory, inside your personal directory.

Important:

- The files `OperatingSystemBase.*` `ComputerSystemBase.*` `ProcessorBase.*` `Simulator.*` `Asserts.*` `Heap.*` `Messages.*` and `messagesTCH.txt` **must not be modified.**

Exercises

0. A MEMADD instruction must be added (but with a different behaviour with respect to the instruction in V0).

   Syntaxis and semantics are:

   ```
   MEMADD registerID memAddress
   ```

   The instruction will add the value contained in the processor register identified by `registerID` (with the same meaning as the one described in the `READ` instruction: accumulator (0), registerA (1), registerB (2)) with the contents of memory address specified by `memAddress`, storing the result in the accumulator register.

1. Implement a function called `ComputerSystem_PrintProgramList()` that **shows the user programs** stored in the vector `programsList`. Take into account that each of its positions points to a C struct and that the first position is reserved for the program implementing the `sipID`, which **is not a user program**.

   To show the information you must use the `ComputerSystem_DebugMessage()` function, using message numbers 101 and 102 (student's messages go in the messages file "`messagesSTD.txt`") and using `INIT` as its second argument. The shown message must be like this:

   ```
   [0] User program list:
   [tab]Program [example1] with arrival time [0]
   ```

```
[tab] …
```

2. Place an invocation to the function implemented in the preceding exercise inside `ComputerSystem_PowerOn()`, **immediately before** initializing the operating system.

3. Invoke the simulator with two arguments identifying the same program, as in this example:

   ```
   ./Simulator --debugSections=HD programVerySimple programVerySimple
   ```

   Is it executed twice? Why? Have a look to the **available system calls** and write a program named `prog-V1-E3` that executes twice when invoked the following way:

   ```
   ./Simulator --debugSections=HD prog-V1-E3 prog-V1-E3
   ```

4. Execute the simulator with a number of user programs in the command line bigger than the capacity of the process table. Which program executes? Why? Make the following modifications to solve the problem:

   a. Modify the function `OperatingSystem_CreateProcess` so it returns to its caller (`OperatingSystem_LongTermScheduler`) the value `NOFREEENTRY` when the first function fails trying to obtain a free entry in the process table.

   b. Modify the function `OperatingSystem_LongTermScheduler` so it distinguishes successful creation and failed creation of a process. In case of failed creation, a message must be displayed in the screen: you must use the `ComputerSystem_DebugMessage()` function, using message number **50 (already defined)** and, `ERROR`, as its second argument. The shown message must be like this:

   ERROR: There are not free entries in the process table for the program [programName]

5. Execute the simulator with a nonexistent user program name in the command line. Analyse what happens. Execute it again, but with a user program that does not contain a number for the process size. Check the result and make the following modifications:

   a. Modify the function `OperatingSystem_LongTermScheduler` so it distinguishes successful creation and failed creation of a process. In case of failed creation, a message must be displayed using the `ComputerSystem_DebugMessage()` function, using message number **51 (already defined)** and, `ERROR`, as its second argument. The shown message must be like this:

   ERROR: Program [programName] is not valid [--- cause of the error ---]

   b. Modify the function `OperatingSystem_CreateProcess` so it returns to its caller (`OperatingSystem_LongTermScheduler`) the value `PROGRAMDOESNOTEXIST` when the first function fails if the program does not exist. The following message must be displayed by `OperatingSystem_LongTermScheduler`:

   ERROR: Program [programName] is not valid [--- it does not exist ---]

c. Do exactly the same for the value `PROGRAMNOTVALID`, returned when the user program does not contain valid integer numbers for the size and priority of the program. Use the following message for both situations:

ERROR: Program [programName] is not valid [--- invalid priority or size ---]

6. Execute the simulator with the program `prog-V1-E6`, that specifies a process size bigger than the maximum size specified by the OS. Why does it work as it works? Try replacing 65 with 50 (a valid value).

```
// prog-V1-E6
65
2
READ 20
ZJUMP 7
SHIFT 1
ZJUMP 2
JUMP -2
TRAP 3
NOP
JUMP -1
TRAP 3
```

Check the result and make the following modifications:

a. Modify the function `OperatingSystem_CreateProcess` so it returns to its caller (`OperatingSystem_LongTermScheduler`) the value `TOOBIGPROCESS` when the first function fails trying to obtain free memory space for the process. The error must be detected when trying to obtain main memory for the program.

b. Modify the function `OperatingSystem_LongTermScheduler` so it distinguishes successful and failed creation of a process. In case of failed creation, a message must be displayed using the `ComputerSystem_DebugMessage()` function, using message number **52 (already defined)** and, `ERROR`, as its second argument. The shown message must be like this:

ERROR: Program [programName] is too big

7. Execute the simulator with a user program (name it `prog-V1-E7`) that specifies a process size smaller than the space needed by its instructions. Check the result and how the `OperatingSystem_LoadProgram()` function deals with such a problem. Finally, make the following modifications:

a. Modify the function `OperatingSystem_CreateProcess` so it returns to its caller (`OperatingSystem_LongTermScheduler`) the value `TOOBIGPROCESS` when the first function detects that the program does not ask for enough space for its instructions.

b. The error message to be displayed would be the same as that of 6-b.

8. Check what happens with processes PIDs when using option `--initialPID=3`. Modify the initialization of `initialPID` (line 82 of original `OperatingSystem.c`) so PID of `SystemIdleProcess` default value is last position of the process table, regardless of its size; in any case, option `--initialPID` can be used. Your code (current and future) should not depend on the way PIDs are assigned to processes.

9. Examine the data structure that contains the list of ready-to-run processes.

    a. Implement a function called `OperatingSystem_PrintReadyToRunQueue()` that shows the contents of the ready-to-run processes queue. To show the information you must use the `ComputerSystem_DebugMessage()` function, using message numbers from 103 and `SHORTTERMSCHEDULE` as its second argument. The shown message must be like this:

    ```
    <tab>[0] Ready-to-run processes queue:
    <tab><tab>[1,0], [3,2], [0,100]
    ```

    where green numbers refer to process identifiers and the black ones refer to their corresponding priority values.

    b. Add an invocation to that new function as the last sentence of the function `OperatingSystem_MoveToTheREADYState()`.

10. We want to see all movements of processes among states. With this purpose in mind:

    a. Paste in `OperatingSystem.c` the following definition:

    ```
    char * statesNames[5]={"NEW","READY","EXECUTING","BLOCKED","EXIT"};
    ```

    b. Locate in the code where processes experience state changes and insert there a call to `ComputerSystem_DebugMessage()` that shows a message like this (message number **53 (already defined)** and section `SYSPROC`), where **`2 - progName`** are the process PID and the name of the file containing the program, respectively:

    ```
    Process [2 - progName] moving from the [READY] state to the [EXECUTING] state
    ```

    If the process is new (there is no previous state), the following message, **54 (already defined),** section `SYSPROC`, must replace message 70:

    ```
    Process [2] created into the [NEW] state, from program[progName]
    ```

11. Modify the short-term scheduling policy.

    a. The new policy will be multilevel queues, with two queues: the one with the highest priority, for the user processes and the other, for the system daemons. To accomplish this task, the number of queues must be 2 and an enumerated date type and an array of queue names must be defined. The initial number of processes in both queues is set to 0.

```
// In OperatingSystem.h
// #define NUMBEROFQUEUES 1
// enum TypeOfReadyToRunProcessQueues { ALLPROCESSESQUEUE };
#define NUMBEROFQUEUES 2
enum TypeOfReadyToRunProcessQueues { USERPROCESSQUEUE, DAEMONSQUEUE};
```

```
// In OperatingSystem.c
// int numberOfReadyToRunProcesses[0]={0};
int numberOfReadyToRunProcesses[NUMBEROFQUEUES]={0,0};

char * queueNames [NUMBEROFQUEUES]={"USER","DAEMONS"};
```

Take into account that memory for 2 queues must be reserved, where before only memory for 1 was obtained.

With the goal of each process knowing the queue it belongs to, the PCB must be augmented with a new field:

```
    int queueID;
```

b. Modify the `OperatingSystem_PrintReadyToRunQueue()` function so it shows the contents of the ready-to-run processes queue like this (message numbers from 108 and section `SHORTTERMSCHEDULE`), taking into account that queue names are obtained from the `queueNames` array:

```
<tab>[0] Ready-to-run processes queues:
<tab><tab>USER: [1,0], [3,2]
<tab><tab>DAEMONS: [0,100]
```

If any of the queues is empty, its name must also appear followed by :

c. Modify all necessary functionality (starting with process creation) so the multilevel queue is correctly used, taking into account that the **short-term scheduler is the only responsible** of deciding the next process to dispatch.

12. Add the new system call `SYSCALL_YIELD` (add it as number 4 in the corresponding enumerated). When the executing process invokes this system call, it will voluntarily relinquish the CPU to a READY process with the same priority and located at the front of its same priority queue. If a process like that did not exist, control transfer would not happen, and the executing process cannot change its state. The function `OperatingSystem_PreemptRunningProcess` probably will help you to implement the preceding functionality. Besides, a call to `ComputerSystem_DebugMessage()` must be made (message **55, already defined,** and `SHORTTERMSCHEDULE` section) showing this information:

```
Process [1 - progName1] will transfer the control of the processor to process [3 - progName2]
```

If the conditions for the operation are not met, nothing must be done and next message must be displayed (number **56, already defined**, section `SHORTTERMSCHEDULE`):

```
Process [1 - progName1] cannot transfer the control of the processor to any process
```

13. Analyze the function `OperatingSystem_SaveContext()`

a. Why is it necessary to save the current value of PC and PSW registers?

b. Is it necessary to save something else?

c. Depending on your answer to questions a) and b), would be necessary any modification of the function `OperatingSystem_RestoreContext()`? Why?

d. Does any of the above changes affect the implementation of any other function or data structure?

e. Depending on your answers to the preceding questions, enter the necessary modification in your code.

14. Check the implementation of `OperatingSystem_TerminateExecutingProcess` and, specially, what happens when the terminated process is `SystemIdleProcess`.

    Modify the function `OperatingSystem_Initialize()` so the simulation ends immediately when the long-term scheduler is incapable of creating any process (the necessary clues to end the simulation are in the implementation of `OperatingSystem_TerminateExecutingProcess()`).

15. We already know what happens when the `HALT` instruction is executed. Write programs containing `OS` and `IRET` instructions. Analyze their behavior. Why do they behave the way they do?

    We want to modify the behavior of some instructions, so they become privileged instructions (privileged instructions should be executed only when the processor's execution mode is also privileged). With this purpose in mind:

    a. Modify the implementation of `HALT`, `OS` and `IRET` so they are only executed in privileged (protected) mode. Otherwise, the processor should raise an exception (that is, an interrupt of the exception type).

    b. Verify that only the OS and daemon processes are executed in privileged mode. When is the protected mode activated and deactivated?