OPERATING SYSTEMS

2024-2025

SIMULATOR OF A MULTIPROGRAMMED COMPUTER SYSTEM WITH CLOCK INTERRUPTS

V2

Introduction

This third version of the simulator experiments a new significant change: clock interrupts appear. Processes state model also changes in a way that they may be moved to the BLOCKED state. We here describe the most important changes.

DESIGN

The operating system

- New or modified data structures:
  - Clock interrupt counter: `numberOfClockInterrupts`
  - Process table: new information to be added.
  - Sleeping processes queue: `sleepingProcessesQueue`, managed as a binary heap, uses the clock interrupt number each process must wake up at as the ordering criteria. It is manipulated like the `readyToRunQueue`.
- Functionality:
  - Short-Term Scheduler (STS).
    - The policy itself remains unchanged but from now on processes coming from the BLOCKED state must be considered.
  - Interrupt handling routines:
    - A new handling routine appears (for clock interrupts), so we have:
      - One for exceptions.
      - A second one for system calls.
      - The new one for clock interrupts.

The processor

The processor acquires the capability of masking interrupts when one is being managed or the system is shutting down.

SIMULATOR OF A MULTIPROGRAMMED COMPUTER SYSTEM WITH CLOCK INTERRUPTS

V2

Initial tasks

Create a copy of your `V1` directory (once you have completed all exercises) and rename it as `V2`. Then, copy the contents of `V2-studentsCode` (eCampus) into your `V2` directory. Finally, execute the following command in your `V2` directory:

```
$ make clean
```

The following exercises must be done working with the contents of the copied set of files.

Exercises

0. To standardize solutions, let us define initial values for registers and fields inside the PCB for new created processes:
   a. General purpose registers must be initialized to 0 (accumulator, A and B registers).
   b. New PCB fields to appear, will be initialized to -1.
1. We are going to add to the system a new interrupt, so that a new interrupt handler is going to be defined.
   a. Set bit 9 of `interruptLines` so it corresponds to the clock interrupt (`CLOCK-INT_BIT=9`) in enumerated INT_BITS in `Processor.h`.
   b. Paste the following code in its corresponding place. Add the necessary function prototype as well:

   ```
   // In OperatingSystem.c  Exercise 1-b of V2
   void OperatingSystem_HandleClockInterrupt(){ return; }
   ```

   c. **<u>Modify whatever it is necessary</u>** to make the function `OperatingSystem_HandleClockInterrupt()` the one handling clock interrupts. Study first how the system deals with other types of interrupts already present in the system.
   d. Modify the clock function `Clock_Update()` to raise a clock interrupt each `intervalBetweenInterrupts` time units, once the clock tic has been incremented by 1. Variable `intervalBetweenInterrupts` has default value 5, but can be modified invoking the simulator using option `--intervalBetweenInterrupts`.
   e. Modify the function `OperatingSystem_HandleClockInterrupt()` so it counts the total number of occurred interrupts (using the variable `numberOfClockInterrupts`) and shows a message like this (`INTERRUPT` section, message number 57, already existing):

```
[12] {0C 009 000} OS 9 0 (PC: 246, Accumulator: 0, PSW: 8082 [M-------X-----Z-])
        [13] Clock interrupt number [2] has occurred
[14] {0D 000 000} IRET 0 0 (PC: 183, Accumulator: 0, PSW: 0082 [--------X-----Z-])
```

2. Think about situations in which a clock interrupt could overlap a previous existing interrupt type handling routine execution.

   Let us modify the processor to mask interrupts if a given PSW bit is set:

   a. Add `INTERRUPT_MASKED_BIT=15` to the enumerated `PSW_BITS` in `Processor.h`.
   b. Add the following code at the end of `Processor_ShowPSW()`, before the `return` sentence, so it shows the value of the new bit:

   ```
   if (Processor_PSW_BitState(INTERRUPT_MASKED_BIT))
         pswmask[tam-INTERRUPT_MASKED_BIT]='M';
   ```

   c. Modify the function `Processor_ManageInterrupts()` so interrupts are not checked **if interrupts are masked**.
   d. Set the PSW `INTERRUPT_MASKED_BIT` bit once the PSW register has been pushed in the system stack (interrupt management function of the processor).

3. Add an invocation to the `OperatingSystem_PrintStatus()` function, implemented in `OperatingSystemBase.c`:
   a. As the last sentence of `OperatingSystem_Initialize()`.
   b. At the end of the code implementing the SYSCALL_YIELD system call **IF** the executing process has been changed.
   c. At the end of the code implementing the SYSCALL_END system call.
   d. At the end of the function handling exceptions.
   e. At the end of the code implementing the Long-Term Scheduler **IF** at least one process has been created.

4. Now, we can avoid showing some messages because of the use of `OperatingSystem_PrintStatus()`. So, comment the invocation to `OperatingSystem_PrintReadyToRunQueue()` we added in exercise V1-9b inside `OperatingSystem_MoveToTheREADYState()`.

5. We are going to add a new system call:
   a. Add a new field to the PCB

   ```
   int whenToWakeUp; // Exercise 5-a of V2
   ```

   b. Paste this code in the specified file:

   ```
   // In OperatingSystem.c Exercise 5-b of V2
   // Heap with blocked processes sorted by when to wakeup
   heapItem *sleepingProcessesQueue;
   ```

`int numberOfSleepingProcesses=0;`

and create a priority queue (`Heap`), pointed by `sleepingProcessesQueue`, for as many processes the system can handle (`PROCESSTABLEMAXSIZE`) in a way similar to the creation of the ready-to-run queues.

c. Define `SLEEPINGQUEUE` inside `OperatingSystem.h` so the compiler will consider the code that depends on the sleeping processes queue definition:

```
#define SLEEPINGQUEUE
```

d. Add a new register to the processor, named `registerD_CPU`, and write the corresponding get and set functions to manipulate it.
e. Modify the implementation of the TRAP instruction, so it copies operand 2 of the instruction inside the new `registerD_CPU`.
f. Add a new system call, SYSCALL_SLEEP=7, that will block the executing process (moving it to the BLOCKED state) and will insert it in the appropriate position (ascending order of `whenToWakeUp`) of the `sleepingProcessesQueue.` Like all system calls, it will be invoked with the TRAP instruction, but the second operand (available inside `registerD_CPU`) will be used to pass an additional value to the system call.

   The value of `whenToWakeUp` (new field inside the PCB) will be obtained by adding a "delay" to the number of clock interrupts that have occurred so far plus an additional unit, to ensure that it will wake up in the future: **whenToWakeUp=delay+numInterrup+1.**

   If the second operand is greater than zero, its value is used as "delay", and if it is less than or equal to zero, the **absolute value** of the accumulator is used as "delay".

   That is, if the value of the second operand is 2 and there have already been 3 clock interruptions, `whenToWakeUp` will be worth 2+3+1=6; the same as if the second operand were <=0, and there were a 2 or -2 in the accumulator: Abs(±2)+3+1.

   The implementation of functions to insert and extract information in/from the `sleepingProcessesQueue` (as it is already done with `readyToRunQueue`) is **strongly recommended**.

g. Add an invocation to the `OperatingSystem_PrintStatus()` function at the end of the code implementing this new system call.
h. Because a new field has been added to the PCB, remember to initialize it appropriately (have a look to exercise 0).

At this point, processes are able to sleep (BLOCKED state) but there is no way for them to wake up, so infinite loops could appear.

6. Modify the function `OperatingSystem_HandleClockInterrupt()` in the following way so the OS checks the `sleepingProcessesQueue` every time a clock interrupt is raised:
   a. If the value of the field `whenToWakeUp` of a process (**or more than one**) of the mentioned queue **equals the number of occurred clock interrupts**, the process will be

unblocked, moved to the READY state and removed from the `sleepingProcessesQueue`.

The implementation of a function to extract information from the `sleepingProcessesQueue` (as it is already done with `readyToRunQueue`) is **strongly recommended**.

b. Once the `sleepingProcessesQueue` has been processed, **if any process has been unblocked**, an invocation to `OperatingSystem_PrintStatus()` must take place.

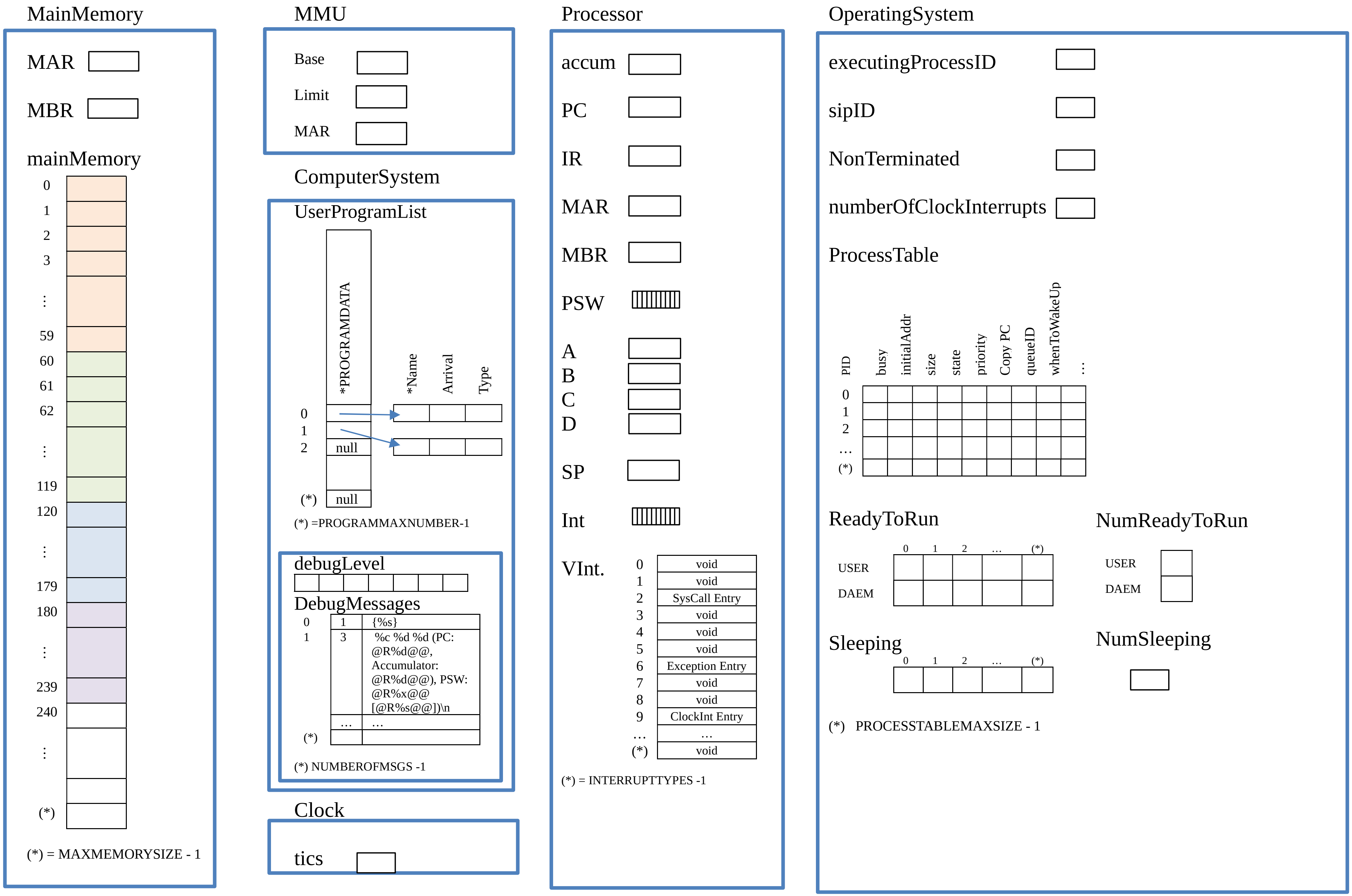
c. In addition to what has been described in the previous paragraph, it will be nec**essary to check if the executing process is the one with the highest priority**. If not, the executing process must give up the processor in favour of the one with the highest priority. Besides, a message must be shown (`SHORTTERMSCHEDULE` section, message number 58, already defined):

```
[27] Process [1 – prNam1] is thrown out of the processor by process [2 – prNam2]
```

d. The `OperatingSystem_PrintStatus()` function must be invoked at the end of `OperatingSystem_HandleClockInterrupt()` if the executing process has changed.

Once correctly finishing the previous exercises, the system status should be printed using `OperatingSystem_PrintStatus()`:

- Each time the Long-Term Scheduler creates new processes.
- Each time a process wakes up.
- Each time the executing process changes.



MainMemory
MAR
MBR
mainMemory
0
1
2
3
59
60
61
62
119
120
179
180
239
240
(*)
(*) = MAXMEMORYSIZE - 1
MMU
Base
Limit
MAR
ComputerSystem
UserProgramList
*PROGRAMDATA
*Name
Arrival
Type
0
1
2
null
(*)
null
(*) =PROGRAMMAXNUMBER-1
debugLevel
DebugMessages
0
1
{%s}
1
3
%c %d %d (PC: @R%d@@, Accumulator: @R%d@@), PSW: @R%x@@ [@R%s@@])\n
…
…
(*)
(*) NUMBEROFMSGS -1
Clock
tics
Processor
accum
PC
IR
MAR
MBR
PSW
A
B
C
D
SP
Int
VInt.
0 void
1 void
2 SysCall Entry
3 void
4 void
5 void
6 Exception Entry
7 void
8 void
9 ClockInt Entry
… …
(*) void
(*) = INTERRUPTTYPES -1
OperatingSystem
executingProcessID
sipID
NonTerminated
numberOfClockInterrupts
ProcessTable
PID
busy
initialAddr
size
state
priority
Copy PC
queueID
whenToWakeUp
0
1
2
…
(*)
ReadyToRun
0 1 2 … (*)
USER
DAEM
NumReadyToRun
USER
DAEM
Sleeping
0 1 2 … (*)
NumSleeping
(*) PROCESSTABLEMAXSIZE - 1