OPERATING SYSTEMS

2024-2025

SIMULATOR OF A MULTIPROGRAMMED COMPUTER SYSTEM

V1

Introduction

Over time, CS became more complex. Accordingly, our second version of the simulator has evolved such as the OS has been supplied with a very important feature: multiprogramming (when all exercises in this version have been completed).

Several changes have been introduced in the components of our CS. We will only pay attention to the important ones.

DESIGN

The processor

The processor has changed. We need a smarter processor in order to implement multiprogramming in the OS. Here is the list of changes:

- Set of registers:
  - New general-purpose registers have been added (`registerA_CPU`, `registerB_CPU` and `registerC_CPU`).
  - A stack pointer register has been also added (`registerSP_CPU`).
  - The PSW register now stores information in a different way. From this version, the PSW register is manipulated as a set of bits. Each bit position has a specific meaning:
    - If `POWEROFF_BIT`$_{th}$ bit position equals 1, the processor must be powered off.
- Interrupt support:
  - The processor is aware of interrupts and processes them accordingly.
    - When an interrupt occurs, a specific bit position of the `interruptLines_CPU` variable will be activated. Each bit position represents an interrupt line, that is, an interrupt type.
  - The address of every interrupt-handling routine (of the OS) will be stored in the interrupt vector table of the processor.
- The processor instruction set also changes:
  - `DIV op1 op2`: calculates the integer division `op1`/`op2`, storing the result in the accumulator register. If `op2` were 0, an exception (an interrupt type) will be raised.
  - `TRAP id`: raises an interrupt of a special type to request a service from the OS. The `id` value specifies the requested service (the processor stores a copy of this `id` value in the `registerC_CPU`).
  - `OS id`: invokes the interrupt handling routine specified by `id`.
  - `IRET`: returns from interrupt management. Similar to the `IRET` instruction of real processors.
  - `MOV op1 op2`: copies the contents of the register specified as op1 to the register specified as op2. These registers may be: accumulator (0), registerA (1) or registerB (2), defined in the source code in an enumerated data type.
  - `INC op1 op2`: the instruction is modified with a second operand that indicates the register over which the operation takes place. Registers can be accumulator (0), registerA (1) or registerB (2). If no second operand is specified, the instruction applies over the accumulator.

- `READ op1 op2` and `WRITE op1 op2`: both instructions are modified, now their second operand indicates the register used for the operation. Registers can be accumulator (0), registerA (1) or registerB (2). The first operand is still a memory address.
- `CALL op1`: instruction to jump to a subroutine pushing the PC of the next instruction to know which instruction to continue with when the subroutine returns. The operand will be the difference between the current address and the initial address of the subroutine to call. That is, the number of addresses to jump (positive numbers denote a forward jump and negative numbers a backward jump).
- `RET`: returns from a subroutine. The pushed PC is popped so it will be used for the next instruction. It corresponds to the widespread RET instruction of processors.

The Memory Management Unit (MMU)

Main memory, as a storing device, does not change. It only increases its capacity, to 300 memory cells, to make room for the operating system code to be executed by the simulated processor.

A different memory size can be specified when invoking the simulator, with option `--memorySize=ValueOfOption`.

A new hardware device, called MMU, appears, interposing between the processor and main memory. Its responsibility consists in translating memory addresses generated by the processor in memory addresses used to index main memory. This device appears because of multiprogramming. As we will see, we need the hardware to collaborate with the OS on isolating the execution of the multiple processes which, usually, will use an independent zone of memory.

Depending on the processor execution mode (protected or user), the MMU will work differently: in protected mode the processor will have access to the whole memory address space.

The computer system

It includes the message subsystem, which stores information related to debugging messages, already used in V0, but in this case, extended and improved.

It is also responsible for storing information about user programs that have been passed from the command line. For each program, it stores its name, the arrival time and that it is a user program (not a daemon, because only user programs are passed from the command line).

The operating system

The OS of this CS is radically different. Multiprogramming forces the OS to define an important amount of data structures, to store information about all the processes, and to implement a large number of functions that define the way the OS controls each process when using the resources of the CS.

- Main data structures:
  - Process table: it is a container for Process Control Blocks (PCBs). Each PCB stores necessary information about a process, such as:
    - Process identifier.
    - Process state.
    - Process priority.
    - Data structuring: pointers to other processes that belong to a given set.
    - Processor state information.
    - Etc.
  - Ready-to-run queue: it is the most significant example of data structuring. This queue is included because the OS must have quick access to the set of processes that are

ready to use the processor.

- Functionality:
  - Long-term scheduler (LTS).
    - Responsible of process admittance.
    - Its main role consists in creating processes from the program list given by the user, from the command line, and processes the OS needs because they play a special role.
  - Process creation.
    - Consists in obtaining a set of resources the process needs to exist, such as:
      - Main memory space for its instructions, data and stack.
      - Loading the program in the allocated memory.
      - Properly initializing its PCB.
      - Initialize the stack at the end of the process address space.
  - Main memory allocation.
    - The OS must allocate an unused main memory portion for each process.
  - Short-term scheduler (STC).
    - It is responsible for the allocation of the processor.
    - At any time, the processor must be allocated to the process with the highest priority.
    - To carry out its work, the STC will use information stored in the ready-to-run queue, in the PCB of the ready-to-run processes and in the PCB of the running process.
  - Dispatcher.
    - It is responsible to give the processor to the process selected by the STC.
      - Changing its state to “EXECUTING”.
      - Replacing the values stored in certain hardware registers with appropriate values for the process.
    - Before giving the processor to the selected process, it must save all relevant information stored in hardware registers for a possible interrupted running process, so it can resume its execution in the future.
  - Interrupt-handling routines.
    - The processor does a basic processing of every interrupt. The processing itself is done by the OS.
    - Therefore, the OS must implement a routine to handle each interrupt type:
      - For exceptions.
      - For system calls.
  - Process termination.
    - There exist several reasons for a process to terminate, but this simulator only considers two:
      - The process produces an exception.
      - The process requests the OS to finish its execution, by means of a specific system call.
    - **When all the user programs have finished**, the OS causes the execution of a `HALT` instruction, so the processor stops and the simulation ends.
- The System Idle Process
  - It is a process that contends for the processor, like the other processes, but with the only purpose of keeping the CPU busy when no other process is ready to use it.
    - Therefore, this process does not do anything special.

```
// System Idle Process
// Executes indefinitely the NOP instruction
4  // Size of the program
100 // Priority of the corresponding process (VERY LOW priority)
ADD 1506 919
NOP
JUMP -1  // Jump an instruction backwards
TRAP 3
```

The clock

A new subsystem appears (`Clock.c` and `Clock.h`), responsible of updating a "*tic*" counter, incrementing it by one each instruction cycle and each time an interrupt is handled.

The clock is updated each time its `Clock_Update()` function is invoked. This happens immediately before invocations to `Processor_FetchInstruction()`, using "wrappers" (see `Makefile`):

```
int __wrap_Processor_FetchInstruction() {
    Clock_Update();
    return __real_Processor_FetchInstruction();
}
```

This also happens before invoking the OS' interrupt logic, `OperatingSystem_InterruptLogic(int)`:

```
void __wrap_OperatingSystem_InterruptLogic(int entryPoint) {
    Clock_Update();
    __real_OperatingSystem_InterruptLogic(entryPoint);
}
```

The `Clock_GetTime()` function is available to get the current value of the "*tic*" counter.

The system buses

Functionality of the address bus is extended, to send addresses from the CPU to the MMU and from the MMU to main memory.

Functionality of the control bus is extended to send control information between the MMU and the CPU and between the MMU and the main memory.

IMPLEMENTATION

The processor

- The new general-purpose registers:

```
int registerA_CPU; // General purpose register
int registerB_CPU; // General purpose register
```

And the way `registerA_CPU` is used for system calls:

```
// Instruction TRAP
case TRAP_INST: Processor_RaiseInterrupt(SYSCALL_BIT);
        registerA_CPU=operand1;
        registerPC_CPU++;
        break;
```

- The new register for the stack pointer `registerSP_CPU`:

```
int registerSP_CPU; // Stack pointer register
```

And the way it is used when invoking a subroutine:

```
// Instruction CALL
case CALL_INST: // Jump to a subroutine
      Processor_CopyInSystemStack(--registerSP_CPU,registerPC_CPU+1);
      registerPC_CPU+=operand1;
      break;
```

- PSW register is now manipulated at bit level. Every bit position (not all of them) has a specific meaning:

```
// Enumerated type that connects bit positions in the PSW register with
// processor events and status
enum PSW_BITS {POWEROFF_BIT=0, ZERO_BIT=1, NEGATIVE_BIT=2, OVERFLOW_BIT=3, EXECU-
TION_MODE_BIT=7};
```

In processor messages, besides showing the PC and accumulator values, the numeric and symbolic values of the PSW are shown.

```
    …
    {0B 000 000} HALT 0 0 (PC: 241, Accumulator: 0, PSW: 0083 [--------X-----ZS])
```

- The processor uses ZERO_BIT for ZJUMP, instead of the accumulator value:

```
// Instruction ZJUMP
case ZJUMP_INST: // Jump if ZERO_BIT
      if (Processor_PSW_BitState(ZERO_BIT))
            registerPC_CPU += operand1;
      else
            registerPC_CPU++;
      break;
```

- The variable `interruptLines_CPU`. Every bit position (not all of them) has a specific meaning regarding interrupts:

```
#define INTERRUPTTYPES 10

// Enumerated type that connects bit positions in the interruptLines_CPU with
// interrupt types
enum INT_BITS { SYSCALL_BIT=2, EXCEPTION_BIT=6};
```

Processor messages include, besides PC and accumulator values, the numeric and symbolic values of the PSW register and the hexadecimal value of `interruptLines_CPU`.

```
…
{0B 000 000} HALT 0 0 (PC: 241, Accumulator: 0, PSW: 0083 [--------X-----ZS], IntLine [0000])
```

- Interrupts support: the OS code is stored in reserved memory positions.

The implementation of the `SO` instruction simulates the execution of OS code, which, as a simplification, IS NOT IMPLEMENTED using processor instructions. A different entry point is used depending on the OS code we want to execute.

```
    …
    SO 6  // EXCEPTION_BIT=6
    IRET  // Return from interrupt
    …
```

The implementation of the `SO` instruction invokes the corresponding OS code using a generic function and specifying the corresponding entry point.

```
…
// Not all operating system code is executed in simulated processor,
// ... but really must do it...
OperatingSystem_InterruptLogic(operand1)
…
```

The interrupt vector table is initialized containing the addresses of code related to each interrupt.

```
void Processor_InitializeInterruptVectorTable(int interruptVectorInitialAddress) {
 int i;
 for (i=0; i< INTERRUPTTYPES;i++) // Initialize all to initial IRET
        interruptVectorTable[i]=interruptVectorInitialAddress-2;

 interruptVectorTable[SYSCALL_BIT]=interruptVectorInitialAddress;//SYSCALL_BIT=2
 interruptVectorTable[EXCEPTION_BIT]=interruptVectorInitialAddress
        + NUMBER_OF_CPU_INSTRUCTIONS_PER_INTERRUPT;          //EXCEPTION_BIT=6
}
```

- Interrupt hardware processing is simple: first of all, the values of the PC and PSW registers are copied to the system stack. Then, the corresponding interrupt-handling routine is executed:

```
// Hardware interrupt processing
void Processor_ManageInterrupts() {

   int i;

   // Interrupts are noted from bit position 1 in the PSW register
   for (i=1;i<INTERRUPTTYPES;i++)
      // If an 'i'-type interrupt is pending
      if (Processor_GetInterruptLineStatus(i)) {
```

```
            // Deactivate interrupt
            Processor_ACKInterrupt(i);
            // Copy PC and PSW registers in the system stack
            Processor_CopyInSystemStack(MAINMEMORYSIZE-1, registerPC_CPU);
            Processor_CopyInSystemStack(MAINMEMORYSIZE-2, registerPSW_CPU);
            // Activate protecte excution mode
            Processor_ActivatePSW_Bit(EXECUTION_MODE_BIT);
            // Call the appropriate OS interrupt-handling routine setting PC register
            registerPC_CPU= interruptVectorTable[i];
            break; // Don't process another interrupt
         }
   }

   // Save in the system stack a given value
   void Processor_CopyInSystemStack(int physicalMemoryAddress, int data) {

      registerMBR_CPU.cell=data;
      registerMAR_CPU=physicalMemoryAddress;
      Buses_write_AddressBus_From_To(CPU, MAINMEMORY);
      Buses_write_DataBus_From_To(CPU, MAINMEMORY);
      registerCTRL_CPU=CTRLWRITE;
      Buses_write_ControlBus_From_To(CPU,MAINMEMORY);
   }
```

The Memory Management Unit (MMU)

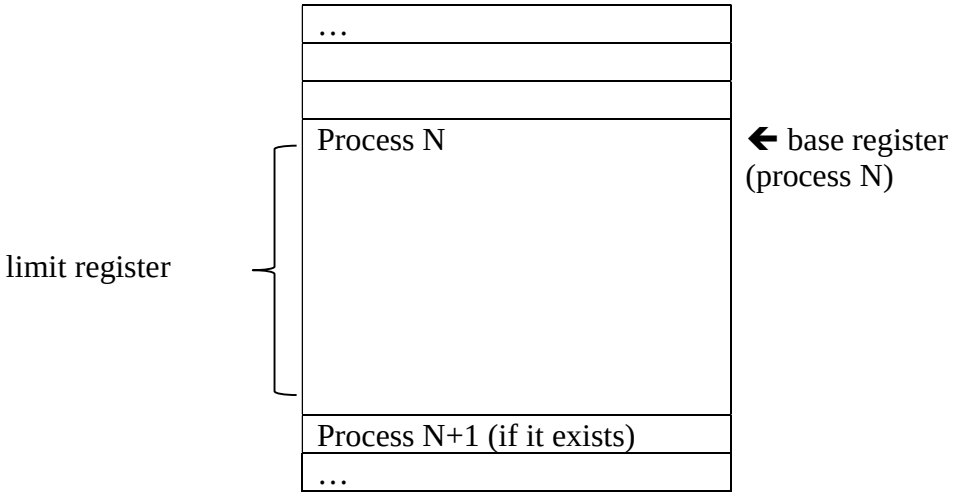
It is implemented in the files `MMU.h` and `MMU.c`. It defines the behavior of a MMU, translating memory address generated by the processor, which are called logical addresses, to addresses used to index main memory, known as physical addresses. Translation takes place both for reading and writing operations, if the processor is not running in protected mode.

```
int registerBase_MMU;
int registerLimit_MMU;
int registerMAR_MMU;
int registerCTRL_MMU;

// Logical address is in registerMAR_MMU. If correct, physical address is produced
// by adding logical address and base register
int MMU_SetCTRL(int ctrl) {
   int firstInvalidAddress;
   registerCTRL_MMU=ctrl&0x3;
   if (Processor_PSW_BitState(EXECUTION_MODE_BIT))
          firstInvalidAddress=MAINMEMORYSIZE; // Protected mode
   else {
          firstInvalidAddress=registerBase_MMU+registerLimit_MMU;    //    Non-Protected
mode
          registerMAR_MMU+=registerBase_MMU;
   }
   switch (registerCTRL_MMU){
   case CTRLREAD
   case CTRLWRITE
          if (registerMAR_MMU < firstInvalidAddress){
                 // Send to the main memory HW the physical address to write in
                 Buses_write_AddressBus_From_To(MMU, MAINMEMORY);
                 // Tell the main memory HW to read
                 // registerCTRL_MMU is CTRLREAD or CTRLWRITE
                 Buses_write_ControlBus_From_To(MMU,MAINMEMORY);
                 // Success
                 registerCTRL_MMU |= CTRL_SUCCESS;
          }
          else { // Fail
                 registerCTRL_MMU |= CTRL_FAIL;
          }
          break;
   default:
          registerCTRL_MMU |= CTRL_FAIL;
          break;
   }
```

```
    // registerCTRL_MMU return value was CTRL_SUCCESS or CTRL_FAIL
    Buses_write_ControlBus_From_To(MMU,CPU);
}
```

To do the translation, when not running in protected mode, the MMU uses two registers, called base and limit registers, that store the appropriate values for the executing process:



The base address always points to the first physical address allocated to the process. The limit register stores the number of memory positions allocated to the process. For example: if the base register stores a 60 and the limit register stores a 35, the logical address 0, generated by the processor, is translated to physical address 0+60=60, that is, the lowest physical address used by the process.

The computer system

Messages of the *Messages* subsystem are now loaded from 2 different files: *messagesTCH.txt* and *messagesSTD.txt*. Messages from the first file have numbers below 100, will be provided by the teachers and must not be modified. Student's messages will be added to the second file.

Access to the messages system is achieved by using a closed hash table.

The function we use to print messages has a new parameter (the first one) used to specify whether a time stamp is printed or not:

```
void ComputerSystem_DebugMessage(int , int, char , ...);
```

The value of this additional parameter is one of the `TimesMessages` enumerated:

```
enum TimedMessages{ NO_TIMED_MESSAGE,TIMED_MESSAGE };
```

Besides, new code appears in *ComputerSystemBase.?* that must not be modified either.

- Data structures
- Programs list

```
// Basic data to collect about every program to be created
// User programs specified in the command line: name of the file, the time of its
//           arrival time to the system (0, by default), and type USERPROGRAM
// Daemon programs of type DAEMONPROGRAM
typedef struct ProgramData {
    char *executableName;
```

```
    unsigned int arrivalTime;
    unsigned int type;
} PROGRAMS_DATA;

PROGRAMS_DATA *programList[PROGRAMSMAXNUMBER];
```

It stores information about programs to be created. For each program, it stores its name, arrival time and whether it is a user or daemon program.

User programs are passed to the Simulator when invoking it, in the command line. The Computer System will include them in the program list.

More programs can be added by means of independent lines in a text file, that may be specified as an option in the command line: `--userProgramsFile=fileName`. Each line must contain the program name and, optionally, its arrival time.

The program list will contain first, programs passed in the command line, and after those, programs specified in the file.

The operating system

This component has suffered the biggest number of changes. It is still a simple OS, but it is necessary to implement an important amount of functionality so it can properly handle several concurrent processes.

- Data structures
    - The process table

```
typedef struct {
      int busy;
      int initialPhysicalAddress;
      int processSize;
      int copyOfSPRegister;
      int state;
      int priority;
      int copyOfPCRegister;
      unsigned int copyOfPSWRegister;
      int programListIndex;
} PCB;

//PCB processTable[PROCESSTABLEMAXSIZE];
PCB * processTable;
```

    - The ready-to-run processes queue. It is a priority queue implemented by using a binary heap (`Heap.c` and `Heap.h`):

```
// Array that contains the identifiers of the READY processes
// heapItem readyToRunQueue[NUMBEROFQUEUES][PROCESSTABLEMAXSIZE];
heapItem * readyToRunQueue[NUMBEROFQUEUES];
int numberOfReadyToRunProcesses[NUMBEROFQUEUES]={0};
```

    - The executing process ID and other important information:

```
// Identifier of the current executing process
int executingProcessID=NOPROCESS;

// Address base for OS code in this version
int OS_address_base; // = PROCESSTABLEMAXSIZE * MAINMEMORYSECTIONSIZE;

// Identifier of the System Idle Process
int sipID;

// Variable containing the number of not terminated user processes
```

```
int numberOfNotTerminatedUserProcesses=0;
```

- Functionality
    - OS Initialization

      It prepares the OS for initial operation.

      It is also responsible of loading into the program list of the CS the programs to create daemon processes. The one corresponding to SystemIdleProcess will always be in the first position of the program list.

    - Long-term scheduler

```
// The LTS is responsible of the admission of new processes in the system.
// Initially, it creates a process from each program specified in the
//          command line and daemons programs
int OperatingSystem_LongTermScheduler() {
  int PID, i, numberOfSuccessfullyCreatedProcesses=0;

  for (i=0; userProgramsList[i]!=NULL && i<USERPROGRAMSMAXNUMBER ; i++) {
     PID=OperatingSystem_CreateProcess(i);
     numberOfSuccessfullyCreatedProcesses++;
     if (programList[i]->type==USERPROGRAM)
           numberOfNotTerminatedUserProcesses++;
     // Move process to the ready state
     OperatingSystem_MoveToTheREADYState(PID);
  }
  // Return the number of succesfully created processes
  return numberOfSuccessfullyCreatedProcesses;
}
```

    - Process creation

```
int OperatingSystem_CreateProcess(int indexOfExecutableProgram) {
…
      FILE *programFile;
      PROGRAMS_DATA *executableProgram=programList[indexOfExecutableProgram];

      // Obtain a process ID
      PID=OperatingSystem_ObtainAnEntryInTheProcessTable();

      // Obtain the memory requirements of the program
      processSize=OperatingSystem_ObtainProgramSize(&programFile,
                                          executableProgram.executableName);

      // Obtain the priority for the process
      priority=OperatingSystem_ObtainPriority(programFile);

      // Obtain enough memory space
      loadingPhysicalAddress=OperatingSystem_ObtainMainMemory(processSize, PID);

      // Load program in the allocated memory
      OperatingSystem_LoadProgram(programFile,loadingPhysicalAddress,processSize);

      // PCB initialization
      OperatingSystem_PCBInitialization(PID, loadingPhysicalAddress, processSize,
                                          priority, indexOfExecutableProgram);

      // Show message "Process [PID] created from program [executableName]\n"
         ComputerSystem_DebugMessage(TIMED_MESSAGE,70,INIT,PID
                                ,executableProgram->executableName);

      return PID;
      }
```

- Process' PCB initialization

```
void OperatingSystem_PCBInitialization(int PID, int initialPhysicalAddress,
                                int processSize, int priority , int processPLIndex) {

    processTable[PID].busy=1;
    processTable[PID].initialPhysicalAddress=initialPhysicalAddress;
    processTable[PID].processSize=processSize;
    processTable[PID].copyOfSPRegister=initialPhysicalAddress+processSize;
    processTable[PID].state=NEW;
    processTable[PID].priority=priority;
    processTable[PID].programListIndex=processPLIndex;
    // Daemons run in protected mode and MMU use real address
    if (programList[processPLIndex]->type == DAEMONPROGRAM) {
            processTable[PID].copyOfPCRegister=initialPhysicalAddress;
            processTable[PID].copyOfPSWRegister=
                                ((unsigned int) 1) << EXECUTION_MODE_BIT;
    }
    else {
            processTable[PID].copyOfPCRegister=0;
            processTable[PID].copyOfPSWRegister=0;
    }
}
```

- Memory allocation for a process

```
// Main memory is assigned in chunks. All chunks are the same size. A process
// always obtains the chunk whose position in memory is equal to the
// processor identifier
int OperatingSystem_ObtainMainMemory(int processSize, int PID) {

    if (processSize>MAINMEMORYSECTIONSIZE)
            return TOOBIGPROCESS;

    return PID*MAINMEMORYSECTIONSIZE;
}
```

- Short-term scheduler

```
// Given that the READY queue is ordered depending on processes priority,
// the STS just selects the process in front of the READY queue
int OperatingSystem_ShortTermScheduler() {

  int selectedProcess;

  selectedProcess=OperatingSystem_ExtractFromReadyToRun();

  return selectedProcess;
}
// Return PID of more priority process in the READY queue
int OperatingSystem_ExtractFromReadyToRun() {

  int selectedProcess=NOPROCESS;

  selectedProcess=Heap_poll(readyToRunQueue[ALLPROCESSESQUEUE],QUEUE_PRIORITY
                                ,&numberOfReadyToRunProcesses[ALLPROCESSESQUEUE]);

  // Return most priority process or NOPROCESS if empty queue
  return selectedProcess;
}
```

- Dispatcher

```
void OperatingSystem_Dispatch (int PID) {

  // The process identified by PID becomes the current executing process
  executingProcessID =PID;
```

```
  // Change the process' state
  processTable[PID].state=EXECUTING;
  // Modify hardware registers with appropriate values for the process PID
  OperatingSystem_RestoreContext (PID);
}
// Modify hardware registers with appropriate values for the process identified by PID
void OperatingSystem_RestoreContext (int PID) {
  // New values for the CPU registers are obtained from the PCB
  Processor_CopyInSystemStack(MAINMEMORYSIZE-1,
                      processTable[PID].copyOfPCRegister);
  Processor_CopyInSystemStack(MAINMEMORYSIZE-2,
                      processTable[PID].copyOfPSWRegister);
  // Same thing for the MMU registers
  MMU_SetBase(processTable[PID].initialPhysicalAddress);
  MMU_SetLimit(processTable[PID].processSize);
}
```

- Entry point to the Operating System for interrupt management (function invoked by the processor):

```
// Implement interrupt logic calling appropriate interrupt handle
void OperatingSystem_InterruptLogic(int entryPoint){
    switch (entryPoint){
           case SYSCALL_BIT: // SYSCALL_BIT=2
                  OperatingSystem_HandleSystemCall();
                  break;
           case EXCEPTION_BIT: // EXCEPTION_BIT=6
                  OperatingSystem_HandleException();
                  break;
    }
}
```

- Interrupt-handling routines

```
void OperatingSystem_HandleException () {
// "Process [executingProcessID] has generated an exception and is terminating\n"
  ComputerSystem_DebugMessage(71,SYSPROC,executingProcessID);

  OperatingSystem_TerminateProcess ();
}

void OperatingSystem_HandleSystemCall () {
    int systemCallID
     // Register A contains the identifier of the issued system call
     systemCallID= Processor_GetRegisterA();

  switch (systemCallID) {
     case SYSCALL_PRINTEXECPID:
           // "Process [executingProcessID] has the processor assigned\n"
           ComputerSystem_DebugMessage(72,SYSPROC,executingProcessID);
           break;
     case SYSCALL_END:
           // "Process [executingProcessID] has requested to terminate\n"
           ComputerSystem_DebugMessage(TIMED_MESSAGE,73,SYSPROC,executingProcessID);
           OperatingSystem_TerminateProcess ();
           break;
  }
}
```

- Process termination

```
// All tasks regarding the removal of the executing process
void OperatingSystem_TerminateExecutingProcess() {

   processTable[executingProcessID].state=EXIT;

   if (executingProcessID==sipID) {
       // finishing sipID, change PC to address of OS HALT instruction
```

```
        Processor_SetSSP(MAINMEMORYSIZE-1);
        Processor_PushInSystemStack(OS_address_base+1);
        Processor_PushInSystemStack(Processor_GetPSW());
        executingProcessID=NOPROCESS;
        ComputerSystem_DebugMessage(TIMED_MESSAGE,99,SHUTDOWN,"The system will shut down
now...\n");
        return; // Don't dispatch any process
    }

    Processor_SetSSP(Processor_GetSSP()+2); // unstack PC and PSW stacked

    if (programList[processTable[executingProcessID].programListIndex]->type==USERPRO-
GRAM)
        // One more user process that has terminated
        numberOfNotTerminatedUserProcesses--;

    if (numberOfNotTerminatedUserProcesses==0) {
        // Simulation must finish, telling sipID to finish
        OperatingSystem_ReadyToShutdown();
    }

    // Select the next process to execute (sipID if no more user processes)
    int selectedProcess=OperatingSystem_ShortTermScheduler();

    // Assign the processor to that process
    OperatingSystem_Dispatch(selectedProcess);
}


void OperatingSystem_ReadyToShutdown(){
    // Simulation must finish (done by modifying the PC of the System Idle Process so
it points to its 'TRAP 3' instruction,
    // located at the last memory position used by that process, and dispatching sipId
(next ShortTermSheduled)
    processTable[sipID].copyOfPCRegister=processTable[sipID].initialPhysicalAd-
dress+processTable[sipID].processSize-1;
    ComputerSystem_DebugMessage(TIMED_MESSAGE,99,SHUTDOWN,"The SystemIdleProcess is
ready to shut down the simulator when dispatched...\n");
}
```

Simulator execution

The simulator is invoked as follows:

```
$ ./Simulator example1 example2 …
```

where `Simulator` is the name of the executable program resulting from compilation, and `example1`, etc. correspond to file names whose content is a user program for the simulator. Files "`SystemIdleProcess`" (process that will execute when no other one is ready to do so) and "`OperatingSystemCode`" (tiny part of the OS code written using the processor instruction set) must also be present in the executing directory.

Simulator invocation admits several options like, for example, `--help`. In the following output, values between square brackets are default values for each option.

```
petrus@ritchie:~$ ./Simulator --help
Use one or more of these options:
        initialPID=ValueOfOption  ["0"]
        endSimulationTime=ValueOfOption  ["-1"]
        numAsserts=ValueOfOption  ["500"]
        assertsFile=ValueOfOption  ["asserts"]
        messagesSTDFile=ValueOfOption  ["messagesSTD.txt"]
        debugSections=ValueOfOption  ["A"]
        daemonsProgramsFile=ValueOfOption  ["DaemonsProgramsFile"]
        userProgramsFile=ValueOfOption  ["UserProgramsFile"]
        memorySize=ValueOfOption  ["300"]
        numProcesses=ValueOfOption  ["4"]
```

```
        generateAsserts
        help
USE: Simulator [--optionX=optionXValue ...] <program1> [arrivalTime] [<program2>
[arrivalTime] .... <program20 [arrivalTime]]
Must have beetwen 1 and 20 program names, or use userProgramsFile option !!!
```

The most important and used option will be `--debugSections`. It is used by the user to specify simulator sections in which it is interested to obtain messages from, during simulation execution. Default value is A (`All`), meaning “show all messages”. Uppercase letters mean coloured messages, lowercase letters, black and white messages. All possible sections are specified in `ComputerSystem.h`.

There are three options related to the assertion system: `numAsserts`,`assertsFile` `generateAsserts`. The assertion system can be used to verify that given simulator components store the correct information at given points of the simulation execution. There is a specific document describing the assertion system.

The remaining options will be introduced when necessary.

V1: Simulator of a Multiprogrammed Computer System

MainMemory

MAR

MBR

CTRL

mainMemory

0	
1	
⋮	
59	
60	
61	
⋮	
119	
120	
⋮	
179	
180	
⋮	
239	
240	
⋮	
(*)	

(*) = MAXMEMORYSIZE - 1

MMU

Base

Limit

MAR

CTRL

ComputerSystem

ProgramList

	*PROGRAMDATA	*Name	Arrival	Type
0				
1				
2	null			
(*)	null			

(*) = PROGRAMMAXNUMBER-1

debugLevel

DebugMessages

0	-1	
1	1	[%s]
2	-1	
3	3	%c %d %d (PC: @R%d@@, Accumulator: @R%d@@), PSW: @R%x@@ [@R%s@@])\n
(*)	-1	

(*) NUMBEROFMSGS -1

Clock

tics

Processor

accum

PC

IR

MAR

MBR

CTRL

PSW

A

B

C

SP

IntLine

VInt.

0	void
1	void
2	SysCall Entry
3	void
4	void
5	void
6	Exception Entry
7	void
8	void
(*)	void

(*) = INTERRUPTTYPES -1

OperatingSystem

executingProcessID

sipID

NonTerminated

ProcessTable

PID	busy	initialAddr	size	state	priority	Copy PC	progListIndex	…
0								
1								
2								
…								
(*)								

ReadyToRun

0		1		…	(*)	
info	count	info	count		info	count

numReadyToRun

(*) PROCESSTABLEMAXSIZE - 1