SISTEMAS OPERATIVOS

CURSO 2019-2020

SIMULADOR DE UN SISTEMA INFORMÁTICO MULTIPROGRAMADO

MANUAL - V1

Introducción

A partir del conocimiento adquirido con el SI Primitivo, continuamos nuestro estudio de los SO con una evolución de aquel Sistema Informático al que, entre otras cosas, habremos dotado de multiprogramación (cuando hayamos completado los ejercicios de esta versión).

Se han producido en el simulador cambios en sus diferentes componentes, que pasamos a detallar. Donde no haya habido cambios o estos hayan sido poco relevantes, no se detallará la naturaleza de los mismos.

DISEÑO

El procesador

El procesador ha cambiado. No sustancialmente pero sí es cierto que necesitamos un procesador más "inteligente" que el de la primera versión. Los cambios introducidos son los siguientes:

- Conjunto de registros:
  - Se ha añadido un registro de propósito general (`registerA_CPU`).
  - Se ha cambiado la forma en que se almacena información en el registro de palabra de estado del procesador (PSW). A partir de esta versión, el registro PSW se manipula como un conjunto de bits. Cada posición de bit tiene un significado concreto:
    - Si la posición de bit `POWEROFF_BIT` está a 1, querrá decir que el procesador se tiene que apagar.
- Soporte para interrupciones:
  - El procesador ya es capaz de reconocer interrupciones y darles el tratamiento hardware correspondiente.
    - La ocurrencia de una interrupción se anotará activando una cierta posición de bit de la variable `interruptLines_CPU`, que simulará la activación de líneas físicas de interrupciones. Se manipula como un conjunto de bits. Cada posición de bit indica un tipo de interrupción.
  - Las direcciones de las rutinas de tratamiento de cada tipo de interrupción se almacenarán en la tabla de vectores de interrupción del procesador.
- El juego de instrucciones del procesador cambia:
  - `DIV op1 op2`: realiza la división entera de `op1` entre `op2` y deja el resultado dentro del registro acumulador. En caso de que `op2` sea 0, eleva una interrupción de tipo excepción.
  - `TRAP id`: produce una interrupción que sirve para solicitar un servicio al SO. El valor `id` permite al SO identificar el servicio solicitado (el procesador almacena el valor `id` en el `registerA_CPU`).
  - `OS id`: realiza una llamada al manejador de la interrupción indicada por el operador `id`.
  - `IRET`: realiza el retorno de un tratamiento de interrupción. Se corresponde con la instrucción `IRET` habitual de los procesadores.

La Unidad de Gestión de Memoria (Memory Management Unit, MMU)

La memoria principal, como dispositivo de almacenamiento, no cambia. Sigue siendo considerada un simple *array* de celdas de memoria, aunque se incrementa el tamaño hasta 300 para que quepa el código del sistema operativo que va a ser ejecutado por el procesador simulado.

Pero lo que sí aparece es un nuevo dispositivo hardware que se interpone entre el procesador y la memoria principal, denominado MMU; que es el encargado de transformar las direcciones de memoria generadas por el procesador, en direcciones de memoria que indexarán la memoria principal. La razón principal de la introducción de este dispositivo es la aparición de la multiprogramación en el SO. Como se verá en el epígrafe correspondiente al SO, necesitamos que el hardware colabore con el SO para conseguir aislar la ejecución de cada uno de los procesos, que deberán ocupar y utilizar en general, zonas de la memoria principal independientes.

El funcionamiento de la MMU variará si el procesador está ejecutándose en modo protegido o no; de forma que, en modo protegido, tendrá acceso a todo el espacio de direcciones de la memoria.

El sistema informático

Además de la información relativa a los mensajes de depuración que ya se usaba en la V0 (el subsistema `Messages` se puede decir que está incluido en el sistema informático); se encarga de guardar información sobre los programas que se van a crear.

Para cada programa almacena su nombre, el instante de llegada y si es un programa de usuario o un daemon.

El sistema operativo

El SO de este SI es el que ha sufrido más cambios. El soporte para múltiples procesos simultáneos obliga al SO a definir una cantidad importante de estructuras de datos, que recogen información sobre cada uno de los procesos, y a implementar un número importante de funciones que definen la forma en que el SO atiende y controla a cada uno de los procesos, en el uso de los diferentes recursos del SI.

- Estructuras de datos fundamentales:
  - Tabla de procesos: esta estructura de datos recoge información básica que el SO necesita registrar para cada proceso. En general, se puede considerar que la tabla de procesos no es más que un contenedor de bloques de control de proceso (Process Control Block, PCB), que recoge información de un proceso como:
    - Identificador del proceso.
    - Estado del proceso.
    - Prioridad del proceso.
    - Pertenencia del proceso a estructuras de datos.
    - Información de estado del procesador.
    - Etc.
  - Cola de procesos listos para ejecución: es interesante que el SO tenga acceso rápido al conjunto de procesos en disposición de utilizar el procesador. Todos esos procesos estarán almacenados en esta estructura de datos.
- Funcionalidad:
  - Planificador a Largo Plazo (PLP) o Long-Term Scheduler (LTS).
    - Responsable de la admisión de procesos.
    - Su papel consiste en solicitar la creación de procesos a partir de los programas que el usuario ha especificado que desear ejecutar con el simulador; y los procesos que crea el sistema por su parte.

- Creación de un proceso.
  - Supone obtener una serie de recursos que el proceso necesita para existir (no necesariamente en el orden que se indica a continuación):
    - Obtener memoria principal para instrucciones y datos.
    - Cargar el programa en la zona de memoria principal asignada.
    - Inicializar adecuadamente el PCB del proceso en la tabla de procesos.
- Asignación de memoria principal a un proceso.
  - Se desea garantizar que todos los procesos obtienen una porción de memoria principal que no está en uso.
- Planificador a Corto Plazo (PCP) o Short-Term Scheduler (STS).
  - Es el responsable de la asignación del procesador.
  - En todo momento debe tener asignado el procesador el proceso más prioritario.
  - Para realizar su tarea, el PCP manipulará la información almacenada en la cola de procesos listos para ejecución y en el PCB de dichos procesos y del proceso en ejecución (el que tiene asignado el procesador).
- Despachador.
  - Es el responsable de asignar el control del procesador al proceso elegido por el PCP.
    - Cambiando el estado de dicho proceso a "EXECUTING".
    - Guardando en ciertos registros del hardware los valores apropiados para este proceso.
  - Antes de realizar la asignación propiamente dicha, tendrá que guardar en el PCB del proceso que pierde el control del procesador toda la información almacenada en registros del hardware que sea de su interés para continuar su ejecución en el futuro, en el estado anterior a dejar de ejecutarse.
- Rutinas de tratamiento de interrupción.
  - El procesador solo sabe dar un tratamiento elemental a las interrupciones, delegando el tratamiento propiamente dicho en el SO.
  - Por tanto, el SO debe implementar una rutina de tratamiento de interrupción para cada tipo de interrupción reconocida por el procesador:
    - Para las excepciones.
    - Para las llamadas al sistema.
- Finalización de un proceso
  - Existen diferentes razones por las que un proceso puede terminar, aunque en este simulador solo hay dos
    - Porque produce una excepción.
    - Porque solicita terminar, vía una llamada al sistema.
  - Cuando **todos** los procesos de **usuario** han finalizado su ejecución, el SO provoca la ejecución de una instrucción `HALT` que, a su vez, provoca la detención del procesador y el final lógico de la simulación.

- El Proceso Inactivo del Sistema
  - Es un proceso que compite por el uso del procesador, como el resto, pero cuyo único propósito es mantener ocupado el procesador cuando ningún proceso de usuario está disponible para ello.
    - Se trata, por tanto, de que no haga nada especial

```
// Proceso Inactivo del Sistema
// Ejecuta la instrucción NOP indefinidamente
4  // Tamanio del programa en posiciones de memoria
100 // Prioridad del proceso generado (MUY baja prioridad)
ADD 1303 617
NOP
JUMP -1  // Saltar una posición de memoria hacia atrás
TRAP 3
```

El reloj

Aparece un nuevo subsistema (`Clock.c` y `Clock.h`), que irá actualizando de forma paralela un contador de "*tics"* en un "*tic*" cada ciclo de instrucción, y otro "*tic*" cada tratamiento de interrupción.

El sistema utilizará la función `Clock_GetTime()` para obtener el "*tic*" de reloj actual cuando lo necesite.

Los buses del sistema

Incrementan la funcionalidad del bus de direcciones para transferir direcciones a la MMU desde la CPU, y a la memoria principal desde la MMU.

Incrementan la funcionalidad del bus de control para transferir información de control entre la MMU y otros componentes del sistema (la CPU, y la memoria principal).

IMPLEMENTACIÓN

EL procesador

- El nuevo registro de propósito general:

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

  Y la forma en que se manipula:

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

- El registro PSW pasa a manipularse bit a bit. Cada posición de bit (no todas) tiene un significado especial:

```
  // 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,
  EXECUTION_MODE_BIT=7};
```

  En los mensajes del procesador, además del PC, y el acumulador, se incluye el valor numérico y simbólico de la PSW.

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

- El procesador utiliza ZERO_BIT para ZJUMP, en vez del valor del acumulador:

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

- La variable `interruptLines_CPU`. Cada posición de bit (no todas) tiene un significado especial relacionado con la ocurrencia de interrupciones:

```
#define INTERRUPTTYPES 10

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

- Soporte de interrupciones: En posiciones de memoria reservadas, se encuentra el código del sistema operativo.

  La implementación de la instrucción SO simula la ejecución del código del sistema operativo que, por simplificar, NO LO IMPLEMENTAMOS en el propio código del procesador de nuestro simulador. Se utiliza un punto de entrada diferente según el código del SO que queramos ejecutar.

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

En la implementación de la instrucción SO, se llama al código que corresponda usando una operación genérica con el punto de entrada que corresponda.

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

Se inicializa el vector de interrupciones que contiene la dirección de acceso al código correspondiente a cada interrupción.

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

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

- El tratamiento hardware de interrupciones es simple: en primer lugar, se copia en la pila del sistema el valor actual de los registros PC y PSW. A continuación, se procede a ejecutar la rutina del SO de tratamiento de la interrupción que se haya producido:

```
// Hardware interruption 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);
}
```

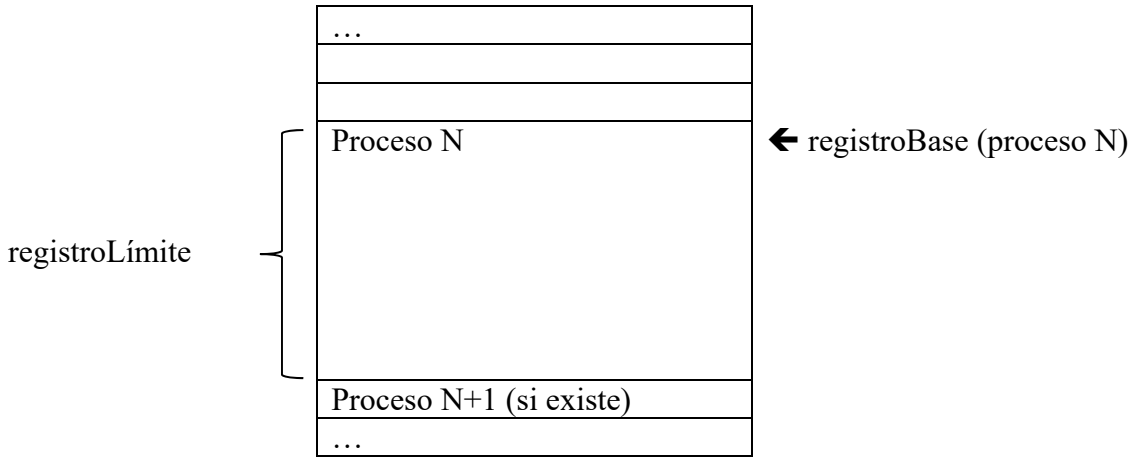
La Unidad de Gestión de Memoria (Memory Management Unit, MMU)

Está implementada en los ficheros `MMU.h` y `MMU.c`. Define el comportamiento básico de una MMU, convirtiendo las direcciones de memoria generadas por el procesador, denominadas direcciones lógicas, en direcciones a utilizar para acceder a la memoria principal, denominadas físicas. La transformación tiene lugar tanto para las operaciones de lectura como las de escritura en memoria principal si el modo de ejecución del procesador no es modo protegido.

```
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) {
   registerCTRL_MMU=ctrl&0x3;
   switch (registerCTRL_MMU){
   case CTRLREAD
      if (Processor_PSW_BitState(EXECUTION_MODE_BIT)){ // Protected mode
          if (registerMAR_MMU < MAINMEMORYSIZE){
                 // 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
                 Buses_write_ControlBus_From_To(MMU,MAINMEMORY);
                 // Success
                 registerCTRL_MMU |= CTRL_SUCCESS;
          }
          else {
                 registerCTRL_MMU |= CTRL_FAIL;
          }
      } else // Non-Protected mode
          if (registerMAR_MMU<registerLimit_MMU) {
                 // Physical address = logical address + base register
                 registerMAR_MMU+=registerBase_MMU;
                 // 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
                 Buses_write_ControlBus_From_To(MMU,MAINMEMORY);
                 // Success
                 registerCTRL_MMU |= CTRL_SUCCESS;
          }
          else
                 registerCTRL_MMU |= CTRL_FAIL;
   break;
   …
   }
   Buses_write_ControlBus_From_To(MMU,CPU);
}
```

Para poder realizar la transformación en caso de ejecutarse en modo no protegido, la MMU se apoya en los valores contenidos en dos registros, base y límite, que toman los valores apropiados para el proceso que tiene el control del procesador en ese momento:



La dirección base siempre apunta a la primera dirección física asignada al proceso. El registro límite registra el número de posiciones de memoria que necesita el proceso. Un ejemplo: si el valor del registro base de un proceso es 60, y el valor de su registro límite es 35, la dirección lógica 0 generada por el procesador se convertirá en la dirección física 0+60=60, es decir, la dirección física de menor valor que utiliza el proceso.

El sistema informático

Almacena la información relativa a los mensajes de depuración (como hacía en la V0). Pero los mensajes de depuración del subsistema *Messages*, pasan a cargarse de 2 ficheros diferentes: *messagesTCH.txt* y *messagesSTD.txt*. Los primeros tienen los números de mensaje inferior a 100, y los proporcionarán los profesores sin posibilidad de ser modificados por el alumno. Los alumnos introducirán sus propios mensajes en el segundo.

La implementación del sistema de acceso a los mensajes por el simulador se hace utilizando una tabla hash cerrada.

Además, aparece una parte en *ComputerSystemBase.?* que incluye la parte de la implementación que NO debe ser modificada por el alumno.

- Estructuras de datos
  - La lista de programas

```
// 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];
```

Se encarga de guardar información sobre los programas que se van a crear. Para cada programa almacena su nombre, el instante de llegada y si es un programa de usuario o un daemon.

El sistema operativo

Aun cuando ha habido cambios en los componentes anteriores, este componente del SI, como es lógico, es el que más cambios ha sufrido. Todavía se trata de un SO sencillo, pero es necesario implementar una cantidad importante de funcionalidad para que pueda controlar adecuadamente un cierto número de procesos simultáneos.

La parte del Sistema Operativo que NO debe ser modificada por el alumno se ha separado en *`OperatingSystemBase.?`*

- Estructuras de datos
  - La tabla de procesos

```
// A PCB contains all of the information about a process that is needed by the OS
typedef struct {
	int busy;
	int initialPhysicalAddress;
	int processSize;
	int state;
	int priority;
	int copyOfPCRegister;
	unsigned int copyOfPSWRegister;
	int programListIndex;
} PCB;

// The process table
PCB processTable[PROCESSTABLEMAXSIZE];
```

  - La cola de procesos listos para ejecución. Es una cola de prioridad implementada como un montículo binario (`Heap.c` y `Heap.h`).

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

  - PID del proceso en ejecución, y otra información necesaria

```
// 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;
```

- Funcionalidad
  - Planificador a Largo Plazo

```
// 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; programsList[i]!=NULL && i<PROGRAMSMAXNUMBER ; 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;
}
```

  - Creación de un proceso

```
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(71,INIT,PID,executableProgram->executableName);

   return PID;
}
```

- Inicialización del PCB de un proceso

```
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].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;
  }
}
```

- Asignación de memoria principal a un proceso

```
// 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;
}
```

- Planificador a Corto Plazo (PCP)

```
// 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,QUEUE_PRIORITY
                    ,&numberOfReadyToRunProcesses);

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

- Despachador

```
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);
}
```

- Entrada al Sistema Operativo del tratamiento de interrupciones invocado desde el procesador.

```
//    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;
      }
}
```

- Rutinas de tratamiento de interrupción

```
void OperatingSystem_HandleException () {
// "Process [executingProcessID] has generated an exception and is terminat-
ing\n"
  ComputerSystem_DebugMessage(72,SYSPROC,executingProcessID,
programList[processTable[executingProcessID].programListIndex]->executableName);

  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(73,SYSPROC,executingProcessID);
       break;
    case SYSCALL_END:
       // "Process [executingProcessID] has requested to terminate\n"
       ComputerSystem_DebugMessage(74,SYSPROC,executingProcessID);
       OperatingSystem_TerminateProcess ();
       break;
  }
}
```

- Finalización de un proceso

```
void OperatingSystem_TerminateProcess () {

  int selectedProcess;

  processTable [executingProcessID].state=EXIT;

..if (programList[executingProcessID]->type==USERPROGRAM)
  // One more process that has terminated
     numberOfNotTerminatedUserProcesses --;

  if (numberOfNotTerminatedUserProcesses ==0) {
…
      // Simulation must finish
      OperatingSystem_ReadyToShutdown();
  }
   // Select the next process to execute (sipID if no more user processes)
   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
   processTable[sipID].copyOfPCRegister=
              processTable[sipID].initialPhysicalAddress+
              processTable[sipID].processSize-1;
}
```

Ejecución del simulador

El simulador se invoca desde línea de comandos, tal y como se puede ver en el siguiente ejemplo:

```
$ ./Simulator ejemplo1 ejemplo2 …
```

Donde `Simulator` es el nombre del programa ejecutable resultante de la compilación. y `ejemplo1`, `ejemplo2`, `etc`. se deben corresponder con los nombres de ficheros que contienen código ejecutable para el simulador. También deben existir en el directorio, los ficheros:

"`SystemIdleProcess`" Proceso que se ejecutará cuando no haya otro proceso que ejecutar.

"`OperatingSystemCode`" El código del sistema operativo.

Se le pueden pasar una serie de opciones como se puede ver ejecutándolo con la opción `--help`. Entre corchetes están las opciones por defecto de cada una de ellas, si son de las que esperan un valor para su asignación.

```
petrus@ritchie:~$ ./Simulator --help
Use one or more of these options:
        --initialPID=ValueOfOption  [0]
        --endSimulationTime=ValueOfOption  [-1]
        --numAsserts=ValueOfOption  [500]
        --assertsFile=ValueOfOption  [asserts]
        --debugSections=ValueOfOption  [A]
        --generateAsserts
        --help
USE: Simulator [--optionX=optionXValue ...] <program1> [arrivalTime] [<program2>
[arrivalTime] .... <program20 [arrivalTime]]
Must have beetwen 1 and 20 program names !!!
```

La opción más importante y utilizada será la de secciones de depuración (opción --debugSections). Con ella se indican las secciones del simulador de las que el usuario está interesado obtener mensajes por pantalla (si se pone alguna en mayúsculas, muestra los mensajes con colores, si son minúsculas, sólo en un color). La opción por defecto es la A (`All`), que indica que se muestran todos los mensajes y además en color, por estar en mayúsculas.

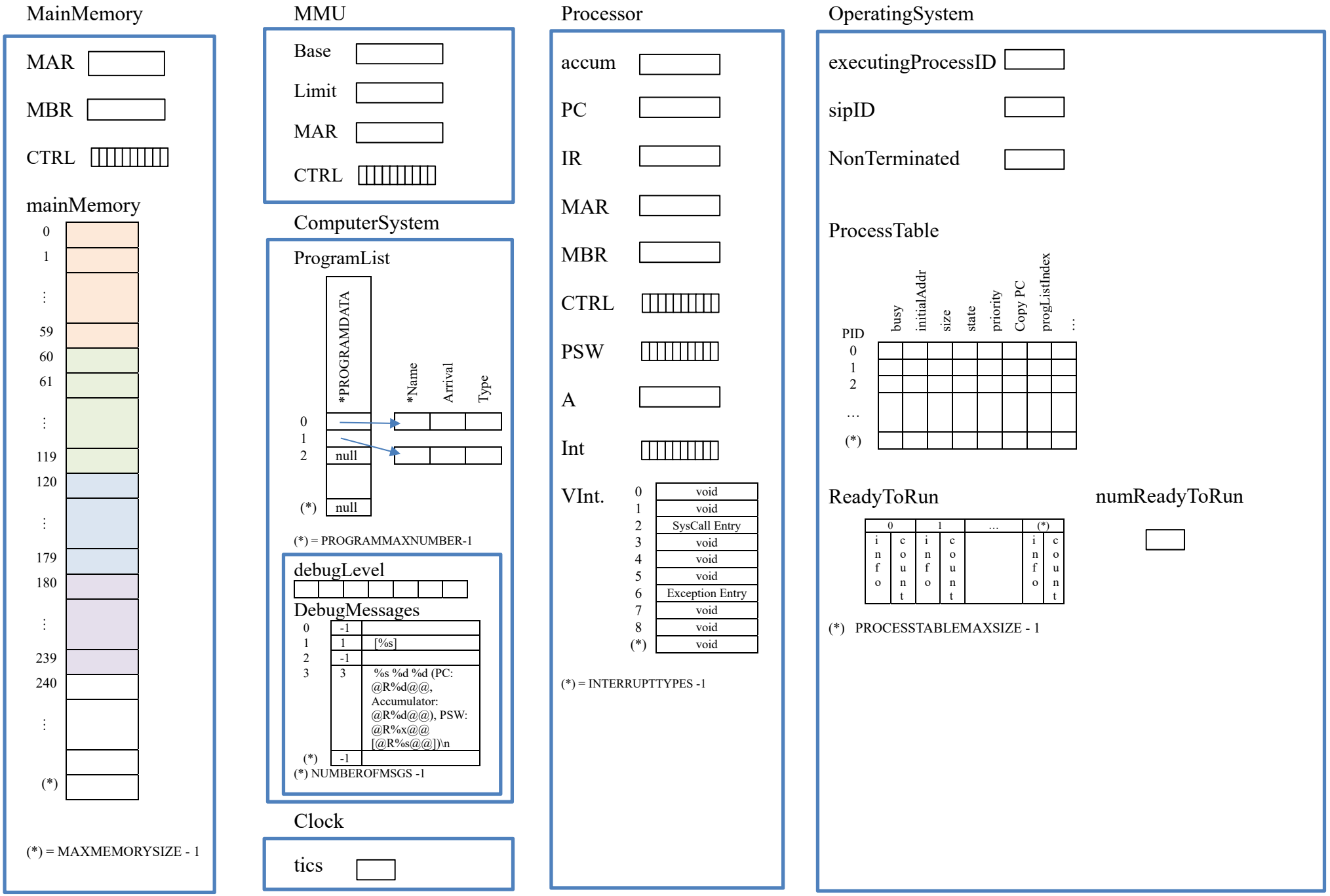
Se pueden ver las posibles secciones en el fichero `ComputerSystem.h`.

Hay tres opciones relacionadas con el sistema de asertos: `numAsserts`,`assertsFile generateAsserts`.

El sistema de asertos (implementado en `Asserts.c` y `Asserts.h`) permite verificar que determinados componentes del Simulador, tienen los valores adecuados en determinados instantes de tiempo.

Para más información sobre el sistema de Asertos, se puede mirar el documento específico que indica su funcionamiento; aunque no es necesaria su comprensión por parte del alumno.

Del resto de opciones se hablará cuando sean utilizadas.



Nota: Se han suprimido los Buses del esquema, pero se siguen utilizando.