SISTEMAS OPERATIVOS

CURSO 2020-2021

SIMULADOR DE UN SISTEMA INFORMÁTICO MULTIPROGRAMADO con gestión de memoria de Particiones Fijas

Manual V4

Introducción

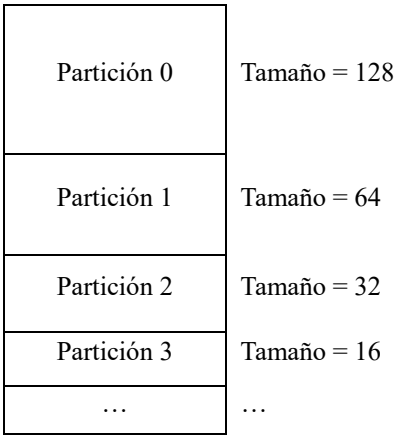
Una vez revisados en la parte teórica de la asignatura los esquemas básicos de gestión de memoria, procedemos a elegir uno de ellos para incorporarlo en nuestro simulador de SI.

Se producen cambios en diferentes componentes del simulador, 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 sistema operativo

El SO debe modificar sustancialmente la forma en la que reparte la memoria entre todos los procesos que la demandan. El esquema de asignación de particiones fijas divide la memoria principal, desde un punto de vista lógico, en particiones (zonas de memoria contiguas) de diferentes tamaños que no varían a lo largo de la vida del sistema:



Cada partición sólo puede estar asignada a un proceso y cada proceso sólo puede estar utilizando una partición. Dado que las particiones existen durante toda la vida del sistema, en un instante de tiempo dado una partición puede estar ocupada o libre.

Cuando se está creando la imagen de un proceso, el sistema operativo debe reservar una partición (libre), con tamaño suficiente para las necesidades de memoria del proceso. Para ello, el SO deberá definir una política (estrategia) de asignación, que permita elegir la partición a asignar al proceso. Por ejemplo:

- Primer ajuste
- Mejor ajuste
- Etc.

Cuando un proceso finaliza su ejecución, el SO debe recuperar la partición asignada y ocupada, marcándola como partición libre para un posible uso posterior por parte de otro proceso.

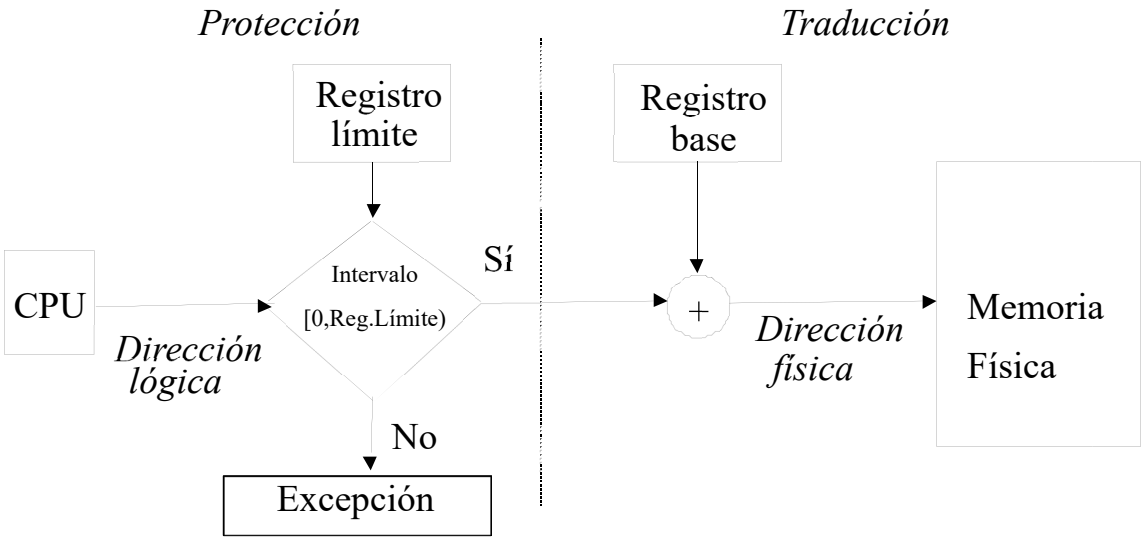
Todas las operaciones anteriores están soportadas por una estructura de datos del SO denominada tabla de particiones, que recoge información sobre todas las particiones en que se divide la memoria, como:

- Dirección física de inicio de la partición.
- Tamaño de la partición.
- Estado de la partición: libre / ocupada.
- Identidad del proceso que está usando la partición (si está ocupada).

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. Pero, lo que sí cambia es la MMU, dispositivo hardware encargado de transformar las direcciones de memoria generadas por el procesador en direcciones de memoria que

indexarán la memoria principal. La razón principal del cambio es la necesidad de que la MMU eleve una excepción cuando la dirección lógica generada por la CPU no supere con éxito la etapa de protección siguiente:



El procesador

El procesador también cambia. No sustancialmente pero sí es cierto que necesitamos un procesador un poco más "inteligente" que el de la versión anterior. Los cambios introducidos son los siguientes:

- Conjunto de registros:
  - Se ha añadido un registro de propósito general (`registerB_CPU`).
- Excepciones:
  - Aparece un nuevo tipo de excepción (hasta ahora división por cero y ejecución de instrucciones protegidas en modo no protegido): la excepción de direccionamiento inválido.
    - Se corresponde con la etiqueta "`Excepción`" de la figura anterior.
  - Será, por tanto, necesario que el procesador distinga los distintos tipos de excepciones:
    - El tipo de la excepción se anotará en el nuevo `registerB_CPU`.

IMPLEMENTACIÓN

El sistema operativo

Es necesario adaptar estructuras de datos y funcionalidad del SO a la nueva estrategia de asignación de memoria.

- Estructuras de datos
  - La tabla de particiones
    - Será un array de `PARTITIONDATA`:

```
typedef struct {
   int initAddress; // Lowest physical address of the partition
   int size; // Size of the partition in memory positions
   int PID; // PID of the process using the partition, or NOPROCESS if free
} PARTITIONDATA;
```

    - Se inicializará a partir de la información almacenada en un fichero de texto, como el

que sigue:

```
128
64
16
32
```

El fichero anterior especifica un particionado de memoria basado en 4 particiones: la primera, de 128 posiciones de memoria, la segunda, de 64, etc.

- Funcionalidad básica
  - Obtención de memoria principal
    - Función `OperatingSystem_ObtainMainMemory()`
      - Esta función tendrá que buscar una partición libre con tamaño suficiente para las necesidades de memoria de un proceso. En caso de éxito, retornará el identificador de la partición encontrada; en caso de fracaso, deberá indicar la causa por la cual no pudo satisfacer las necesidades del proceso.
  - Liberación de memoria principal
    - Función `OperatingSystem_ReleaseMainMemory()`
      - Cuando un proceso finaliza su ejecución, el SO recupera (libera) la memoria que utilizaba. Esta función deberá actualizar la tabla de particiones para liberar la partición que había sido ocupada por el proceso que termina.

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

Deberá elevar una excepción de direccionamiento inválido cuando la dirección lógica generada por el procesador no supere las comprobaciones a realizar en la etapa de protección. Teniendo en cuenta que las direcciones lógicas válidas generadas por el procesador durante la ejecución de un proceso P cualquiera son las comprendidas en el intervalo [0, tamañoDelProceso-1], la MMU elevará una excepción siempre que la dirección lógica esté fuera de dicho intervalo.

El procesador

- El nuevo registro de propósito general:

```
int registerB_CPU; // Registro de propósito general
```

Y la forma en que se manipula:

```
void Processor_RaiseException(int typeOfException) {
   Processor_RaiseInterrupt(EXCEPTION_BIT);
   registerB_CPU=typeOfException;
}
```

SIMULADOR DE UN SISTEMA INFORMÁTICO MULTIPROGRAMADO con gestión de memoria de Particiones Fijas

Tareas V4

Tareas iniciales

Saca un duplicado de tu directorio `V3` (una vez completada esta versión) denominándolo `V4`. El trabajo a realizar en los ejercicios siguientes se desarrollará sobre la copia indicada de los ficheros contenidos en el directorio `V4`, dentro de tu directorio personal. Copia también en dicho directorio `V4` los recursos disponibles en `V4-studentsCode-2020-21` descargados del Campus Virtual.

Ejercicios

1. Vamos a añadir una nueva excepción y de paso, cambiar el modo de elevar excepciones.

    a. Añade un nuevo registro a tu CPU:

    ```
    int registerB_CPU; // Another general purpose register Exercise 1-a of V4
    ```

    b. Añade también el siguiente enumerado al fichero `Processor.h`. Dicho enumerado contiene los diferentes tipos de excepción que se van a poder producir:

    ```
    enum EXCEPTIONS {DIVISIONBYZERO, INVALIDPROCESSORMODE, INVALIDADDRESS, INVALIDINSTRUCTION};
    ```

    c. En `ProcessorBase.c` y `ProcessorBase.h` ya están el código y la declaración de la función `Processor_RaiseException`. Dicha función deberá ser invocada cada vez que se produzca una excepción, pasándole como argumento el tipo de excepción ocurrida (alguno de los valores del enumerado anterior).

    ```
    // Function to raise an exception.
    void Processor_RaiseException(int typeOfException) {
           Processor_RaiseInterrupt(EXCEPTION_BIT);
    #ifdef MULTIPLE_EXCEPTIONS
           registerB_CPU=typeOfException;
    #endif
    }
    ```

    Para que se compile y poder utilizarla adecuadamente, añade esta línea a tu `Processor.h`

    ```
    #define MULTIPLE_EXCEPTIONS
    ```

    A partir de ahora las excepciones se elevan de otra forma, por lo que **las excepciones existentes con anterioridad deben adaptarse** a la nueva forma de elevar excepciones.

d. Modifica el código de tu MMU para que tenga en cuenta que el rango de direcciones lógicas válidas para el esquema de particiones fijas está comprendido en el intervalo `[0, tamañoProceso-1]`. Si la dirección lógica está fuera de ese intervalo, la MMU tendrá que elevar una excepción del tipo `INVALIDADDRESS`.

2. Modifica la función `OperatingSystem_HandleException` para que muestre por pantalla un mensaje de error personalizado (número de mensaje 140, sección `INTERRUPT`) que dependa del tipo de excepción ocurrida, con el aspecto siguiente:

```
[24]_Process_[1_-_NomProgPid1]_has_caused_an_exception_(invalid_address)_and_is_being_terminated
[31]_Process_[3_-_NomProgPid3]_has_caused_an_exception_(invalid_processor_mode)_and_is_being_terminated
[37]_Process_[2_-_NomProgPid2]_has_caused_an_exception_(division_by_zero)_and_is_being_terminated
```

3. Hasta ahora, las instrucciones no válidas se trataban como si fuesen la instrucción NOP. Vamos a cambiarlo:

   a. Modifica el código del procesador para que detecte instrucciones que le son desconocidas. En dicho caso, deberá elevar una excepción del tipo `INVALIDINSTRUCTION`, y no hacer nada más.

   b. Modifica el código del SO para que muestre por pantalla el mensaje de error 140 (sección `INTERRUPT`) con el aspecto siguiente cuando el procesador le advierta de la ocurrencia de una excepción de tipo `INVALIDINSTRUCTION`. Dado que es una excepción, el proceso que la produce será obligado a finalizar su ejecución, tal y como ocurre con el resto de excepciones.

```
[71]_Process_[2_-_NomProgPid2]_has_caused_an_exception_(invalid_instruction)_and_is_being_terminated
```

4. Hasta ahora tampoco tenía ninguna consecuencia que un proceso hiciese una llamada a sistema inexistente.

   a. Vamos a modificar el código del SO para que muestre por pantalla un mensaje de error (número de error 141, sección `INTERRUPT`) cuando una llamada al sistema realizada por un proceso de usuario no sea reconocida como llamada al sistema válida. El mensaje debe tener el siguiente aspecto:

```
[79]_Process_[1_-_NomProgPid1]_has_made_an_invalid_system_call_(777)_and_is_being_terminated
```

   Donde el valor entero **777** se corresponde con el identificador numérico de llamada al sistema realizada por el usuario y que no ha sido reconocido como el de una llamada al sistema válida (`TRAP 777`).

   b. Además, el proceso que produce tal error será obligado a finalizar su ejecución.
   c. Añade como última instrucción del tratamiento de una llamada a sistema no válida, una llamada a `OperatingSystem_PrintStatus()` puesto que cambian algunos estados de procesos.

5. La tabla de particiones será un array de `PARTITIONDATA`, cuya definición y declaración ya están incluidas en `OperatingSystemBase.h` y `OperatingSystemBase.c`:

```
typedef struct {
     int initAddress; // Lowest physical address of the partition
     int size; // Size of the partition in memory positions
     int PID; // PID of the process using the partition, or NOPROCESS if free
} PARTITIONDATA;

// Partition table
PARTITIONDATA partitionsTable[PARTITIONTABLEMAXSIZE];
```

Siendo `PARTITIONTABLEMAXSIZE` el doble que el número máximo de procesos. Es decir, se pueden definir hasta el doble de particiones de memoria que procesos soporta el simulador.

Para inicializar la tabla de particiones de SO a partir de los datos almacenados en el fichero "`MemConfig`" se utiliza la función `OperatingSystem_InitializePartitionTable()` que también está en `OperatingSystemBase.c` que además, **devuelve** el número de particiones leídas. Si hay más líneas en el fichero que particiones en el sistema, sólo utiliza las primeras líneas.

También se muestra la tabla de particiones inicial.

Para que se compile el código de dicha función y definir el nombre del fichero con la configuración de la memoria, pega el código siguiente en `OperatingSystem.h`:

```
// Partitions configuration file name definition
#define MEMCONFIG "MemConfig"  // in OperatingSystem.h
```

La función `OperatingSystem_InitializePartitionTable()` deberá ser invocada desde la función `OperatingSystem_Initialize()`, antes de que el SO considere la creación de proceso alguno.

6. Se modifica la política de asignación de memoria.

   a. Modifica la función `OperatingSystem_ObtainMainMemory()` para que elija una partición para el proceso siguiendo la política (estrategia) del **mejor ajuste**; si hubiese varias, se debe elegir la que empiece en una dirección de memoria más baja.

   b. **Siempre que se intente obtener memoria** para un proceso, se debe mostrar un mensaje (número de mensaje 142, sección `SYSMEM`) con el aspecto siguiente:

   ```
   [6]_Process_[1_-_NomProgPid1]_requests_[20]_memory_positions
   ```

   c. **Si la búsqueda tiene éxito**: la función retornará el identificador (número) de partición encontrada. **IMPORTANTE**: dado que la naturaleza del valor de retorno cambia, tendrás que modificar también la función `OperatingSystem_CreateProcess()`. Además, **si el proceso se crea correctamente**, se mostrará un mensaje (número de mensaje 143, sección `SYSMEM`) con el aspecto siguiente:

   ```
   [6]_Partition_[2:_192_->_32]_has_been_assigned_to_process_[1_-_NomProgPid1]
   ```

   Donde **`2`** es el número de partición, **`192`** es la dirección física de comienzo de la partición y **`32`** es el tamaño de la misma.

d. Si la búsqueda fracasa, lo puede hacer por dos razones:

   i. El proceso solicita una cantidad de memoria superior a la de cualquier partición. La función deberá retornar un error `TOOBIGPROCESS`

   ii. El proceso cabe en alguna partición, pero todas las particiones con tamaño suficiente están ocupadas. La función deberá retornar un error `MEMORYFULL`. Para este último caso, añade la siguiente definición:

```
#define MEMORYFULL -5  // In OperatingSystem.h
```

Además, se mostrará un mensaje (número 144, sección `ERROR`) con el aspecto siguiente:

```
[1]_ERROR:_A_process_could_not_be_created_from_program_[acceptableSizeExample]
_because_an_appropriate_partition_is_not_available
```

7. En `OperatingSystemBase.c` está implementada una función que se utiliza para mostrar la tabla de particiones de memoria: `OperatingSystem_ShowPartitionTable(char *)`

   El parámetro que se le tiene que pasar en este ejercicio es una de las cadenas de caracteres siguientes (recuerda sustituir cada guion bajo por un espacio):

```
before_allocating_memory
after_allocating_memory
```

   La función deberá ser invocada antes y después de la operación de obtención de memoria, con el fin de mostrar los cambios que sufre la tabla de particiones como resultado de la realización de las operaciones indicadas.

   El resultado de la ejecución de llamada a la función tiene este aspecto:

```
[125] Main memory state (after allocating memory):
  [0] [0 -> 4][0 - ProgramNameInPartition0]
  [1] [4 -> 12][AVAILABLE]
  [2] [16 -> 16][AVAILABLE]
  [3] [32 -> 96][1 - ProgramNameInPartition3]
  [4] [128 -> 64][AVAILABLE]
  [5] [192 -> 16][AVAILABLE]
  [6] [208 -> 32][3 - ProgramNameInPartition6]
```

   El primer valor entre corchetes indica el número de partición; el segundo, las características de la partición; y el tercero, si la partición está libre u ocupada (en este caso, figura el PID del proceso que la ocupa y también el nombre del programa).

MUY IMPORTANTE:

Solo se deben mostrar los mensajes de la tabla de particiones y reservar la partición de memoria elegida si los procesos se crean correctamente. Si no se crea el proceso, solo saldrá el mensaje con la solicitud de posiciones de memoria.

8. Añade una función `OperatingSystem_ReleaseMainMemory()` que será invocada durante la finalización de un proceso y que deberá liberar la partición de memoria ocupada por éste. La función deberá mostrar un mensaje como el siguiente (número 145, sección `SYSMEM`):

   ```
   [123]_Partition_[0:_0_->_128]_used_by_process_[1_-_NomProgPid1]_has_been_released
   ```

   Además, utilizando la función que muestra la tabla de particiones del ejercicio anterior, deberá ser invocada antes y después de operaciones de liberación de memoria, con el fin de mostrar los cambios que sufre la tabla de particiones como resultado de la realización de las operaciones indicadas.

   El parámetro que se le tiene que pasar en este ejercicio es una de las cadenas de caracteres siguientes (recuerda sustituir cada guion bajo por un espacio):

   ```
   before_releasing_memory
   after_releasing_memory
   ```

   Y la primera línea de la salida de la ejecución incluiría los nuevos mensajes:

   ```
   [234] Main memory state (before releasing memory):
   ```

Escuela Ingeniería Informática de Oviedo. Sistemas Operativos: Simulador V4 Final

MainMemory

MAR

MBR

mainMemory

0 1 2 3 4 5 6 7 8 … (**) … (*)

(**) =PartitionsSizeSum -1

(*) = MAXMEMORYSIZE - 1

Clock

tics

MMU

Base

Limit

MAR

ComputerSystem

ProgramList

*PROGRAMDATA | *Name | Arrival | Type

0 1 2 null … (*) null

(*) = PROGRAMMAXNUMBER-1

Arrival
(index in programList)

0 1 2 … (*)

(*) = PROGRAMMAXNUMBER-1

NumArrival

Messages Subsystem
….

Processor

accum

PC

IR

MAR

MBR

PSW

A

B

Int

VInt.

0	null
1	null
2	SysCall Entry
3	null
4	null
5	null
6	Exception Entry
7	null
8	null
9	ClockInt Entry
…	
(*)	null

(*) = INTERRUPTTYPES -1

OperatingSystem

executingProcessID

sipID

NonTerminated

numberOfClockInterrupts

processTable

PID | busy | initialAddr | size | state | priority | Copy PC | programListIndex | queueID | whenToWakeUp | …

0 1 2 … *

partitionsTable

initAddress | size | PID

0 1 2 … (**)

(**) 2*PROCESSTABLEMAXSIZE – 1

ReadyToRun (PID)

0 1 2 … (*)

USER

DAEM

NumReadyToRun

USER

DAEM

Sleeping (PID)

0 1 2 … (*)

NumSleeping

(*) PROCESSTABLEMAXSIZE – 1