OPERATING SYSTEMS

2024-2025

SIMULATOR OF A MULTIPROGRAMMED COMPUTER SYSTEM with a variable partitioning memory management system

V4

Introduction

Once the theoretical memory management concepts have been reviewed, we are going to choose one of the possible schemes for our CS simulator.

Again, the simulator experiences several changes that affect several of its components. We will only give details about the relevant ones.

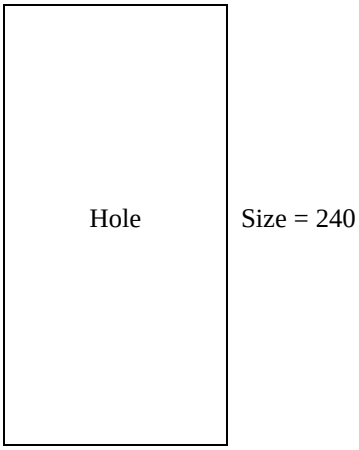
DESIGN

The operating system

The OS must substantially change the way it allocates memory to processes demanding it. A variable partitioning scheme makes a logical division of main memory in regions of contiguous memory positions of two different types:

- Partitions: regions occupied by processes.
- Holes: free regions.

When the computer boots, the OS allocates for its instructions and data a given region, and the remaining main memory becomes a unique big hole:

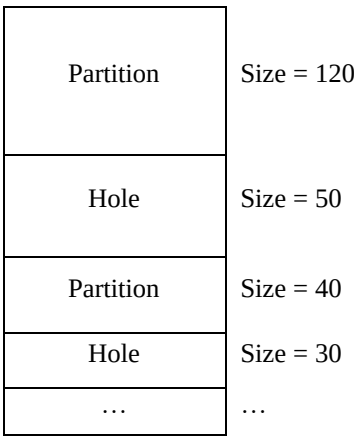


When memory must be allocated to a process, the OS searches among the existing holes for one big enough for the process and assigns the strictly necessary space. Usually, the hole's size is bigger than the process' size, so, after the assignment, a smaller hole will be generated for future needs. The OS must define an assignment policy (strategy) which determines the most suitable hole for the process. For example:

- First fit.
- Best fit.
- Etc.

When a process finalizes its execution, the OS must recover the occupied space, labeling it as a hole, for future use for another process. In case this new hole is adjacent to other existing hole(s), it will be necessary to coalesce them into a single hole, whose size will be equal to the size of all adjacent hole.

After several assignment and releasing operations, main memory arrangement has an aspect similar to this:

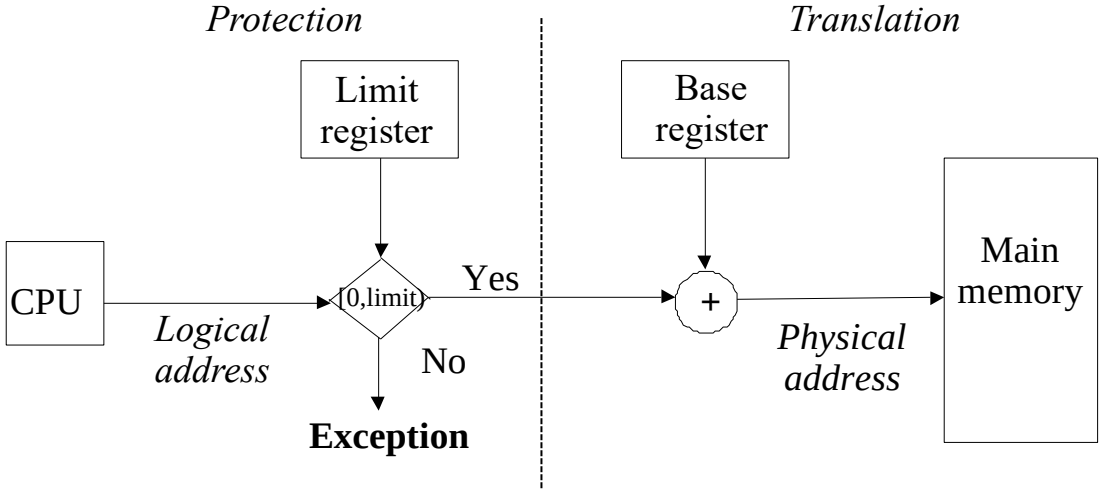


All of the mentioned operations are supported by a specific data structure of the OS called partitions and holes table. This table stores information about partitions like:

- Initial physical address of the partition/hole.
- Size.
- Partition/hole.
- Identity of the process using the partition (if occupied).

The Memory Management Unit (MMU)

Main memory, as a storage device, does not change. But the MMU changes. The MMU is the responsible of the translation of memory addresses generated by the processor, called logical addresses, to addresses used to index main memory, called physical addresses. We now need that the MMU raises an exception whenever the logical address does not pass the protection mechanism shown in the following figure:



The processor

The processor also changes. We need a processor a little bit more intelligent. The new features are:

- Exceptions:
  - A new type of exception appears (until now, the division-by-zero and the execution of privileged instructions in no-privileged mode): invalid address exception.

        - It is the exception labeled in the previous figure.
    - So, from now on, it will be necessary for the processor to differentiate both types (or any number of types) of exceptions:
        - The type of the raised exception will be written in the new `registerD_CPU`.

IMPLEMENTATION

The operating system

It is necessary to add new data structures and functionality to the OS due to the new memory management scheme.

- Data structures:
    - The partition table:
        - It will be an array of `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 HOLE if free
} PARTITIONDATA;
```

- Basic functionality:
    - Main memory allocation:
        - Function `OperatingSystem_ObtainMainMemory()`
            - This function is responsible of searching for an available hole whose size is enough for a process. If the search succeeds, the function will return the identifier of the found partition; if it fails, the function must indicate the reason for which it could not satisfy the needs of the process.
    - Main memory release:
        - Function `OperatingSystem_ReleaseMainMemory()`
            - When a process finishes its execution, the OS recovers (releases) the main memory the process was using. This function will update the partitions and holes table in order to release the partition the process had occupied, coalescing holes, if necessary.

The Memory Management Unit (MMU)

The MMU will raise an invalid address exception whenever the logical address generated by the processor does not pass the protection stage (see figure above). Given that the expected addresses generated by the processor during the execution of a process P must be in the interval [0, sizeOfTheProcess-1], the MMU will raise an exception every time the logical address is out of that interval.

The processor

- The new (and additional) way we will use system register D, to store the type of the raised exception:

```
int registerD_CPU; // System register
```

And the way it will be manipulated:

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

SIMULATOR OF A MULTIPROGRAMMED COMPUTER SYSTEM with a variable partitioning memory management system

V4

Initial tasks

Create a copy of your `V3` directory and rename it as `V4`. The following exercises must be done working with the contents of the copied set of files. Then, copy the contents of `V4-studentsCode-2024-25` (eCampus) into your `V4` directory. Finally, execute the following command in your `V4` directory:

```
$ make clean
```

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

Exercises

1. We are going to add a new exception and, at the same time, change the way they are raised:

   a. Add the following enumerate type to the file `Processor.h`. This enumerate contains the different types of exceptions that can occur:

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

   b. `ProcessorBase.c` and `ProcessorBase.h` already contain the code of the function `Processor_RaiseException`. This function must be invoked every time an exception is raised. The argument for the function will be the type of the occurred exception (one of the above values).

```
// Function to raise an exception
void Processor_RaiseException(int typeOfException) {
	Processor_RaiseInterrupt(EXCEPTION_BIT);
	registerD_CPU=typeOfException;
}
```

   This is the new way to raise exceptions, so you have **to modify your code to adapt the way you were raising exceptions** so far.

   c. Modify the code of the MMU so it validates the logical address received from the CPU. Remember valid logical addresses are in the interval *[0, sizeOfTheProcess-1]*. If the logical address is out of that interval, the MMU must raise an `INVALIDADDRESS` exception. Have in mind that, if the processor is executing

in privileged mode, `INVALIDADDRESS` exceptions must be also raised considering the corresponding address range.

2. Modify the function `OperatingSystem_HandleException` so it shows the corresponding error message (`INTERRUPT` section, message number 140) depending on the type of exception raised:

```
[24] Process [1 - ProcessName1] has caused an exception (invalid address) and is being terminated
[31] Process [3 - ProcessName3] has caused an exception (invalid processor mode) and is being terminated
[37] Process [2 - ProcessName2] has caused an exception (division by zero) and is being terminated
```

3. So far, invalid instructions have been processed as the NOP instruction. It is time to change that:

    a. Modify the code of the processor so it detects invalid (unknown) instructions. In this case, it must raise an `INVALIDINSTRUCTION` exception and do nothing else.

    b. Modify the code of the OS to handle this new type of exception. As before, the process must be terminated, and an appropriate error message (`INTERRUPT` section, message number 40, already existing) must be displayed:

```
[71] Process [2 - ProcessName2] has caused an exception (invalid instruction) and is being terminated
```

4. So far, requesting a non-existing system call has not had any consequences.

    a. Modify the code of the OS to show an error message (`INTERRUPT` section, message number 41) when a system call made by a process is not recognized as a valid system call:

```
[79] Process [1 - ProcessName1] has made an invalid system call (777) and is being terminated
```

    Where the value 777 corresponds to the system call identifier the process has used and not recognized by the operating system as a valid system call (`TRAP 777`).

    b. Besides, the process that caused this error must be terminated.

    c. To end managing this situation, invoke `OperatingSystem_PrintStatus()`, because some processes are expected to get their states modified.

5. The partitions and holes table will be an array of `PARTITIONDATA`, already defined in `OperatingSystemBase.h` and `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 HOLE if free
} PARTITIONDATA

// Partition table
PARTITIONDATA * partitionsAndHolesTable;
```

And it is used as if it were an array of size `PARTITIONSANDHOLESTABLEMAXSIZE` which is twice that of the maximum number of processes. In other words, you can define up to twice as many memory partitions as the processes the simulator supports.

The maximum number of partitions and holes could be greater, in case option `--partitionsAndHolesTableSize=ValueOfOption` is used when invoking the simulator.

The partitions and holes table is initialized by invoking `OperatingSystem_InitializePartitionsAndHolesTable(int)`, defined in `OperatingSystemBase.c`. The function initializes the array of partitions and holes, initially with a unique hole whose size must be the available memory, passed as a parameter.

The function also prints the initial status of the partitions and holes table.

a. To properly compile that function and to define the name of the memory configuration file, paste the following in `OperatingSystem.h`:

```
// Partitions and holes configuration file name definition
#define MEMCONFIG // in OperatingSystem.h
```

b. The function `OperatingSystem_InitializePartitionsAndHolesTable ()` must be invoked from inside `OperatingSystem_Initialize()` before the Operating System considers the creation of any process.

6. The memory management policy is being modified:

a. Modify the function `OperatingSystem_ObtainMainMemory()` so it selects a hole for the process using the **best fit** strategy. If there were more than one eligible holes, the chosen one should be that using the lowest physical addresses.

b. A message (`SYSMEM` section, existing message number 42) like the following must be displayed:

```
[6] Process [1 - ProcessName1] requests [20] memory positions
```

c. If the **search succeeds**, the mentioned function will return the identifier (number) of the found hole. **IMPORTANT**: given that the semantics of the returned value changes, it will be also necessary to change the function `OperatingSystem_CreateProcess()`. Besides, **if the process is successfully created**, a message like the following (`SYSMEM` section, existing message number 43) must be displayed:

```
[6] Partition [2: 192-> 32] has been assigned to process [1 - ProcessName1]
```

Where **2** is the partition number, **192** is the initial physical address of the partition and **32** is its size.

If the space assignment generates a new hole, information about this fact must be

printed (SYSMEM section, existing message number 44):

```
[6] New hole [3: 224 -> 12] has been created after assigning memory to
process [1 - NomProgPid1]
```

d. If the search for a partition finishes with failure, one of the following problems has occurred:

   i. The process requires more memory than the size of biggest hole ever. The function must return a TOOBIGPROCESS error.

   ii. The process fits in memory, but all existing holes are not big enough. The function must return a MEMORYFULL error (add the following definition):

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

   Besides, a message like the following must be displayed (ERROR section, existing message number 39):

[12] ERROR: a process could not be created from program [acceptableSizeProgramName] because an appropriate partition is not available

7. Add as many invocations as necessary to the function OperatingSystem_ShowPartitionsAndHolesTable(char *) (already implemented in OperatingSystemBase.c) to show the main memory status before and after memory allocation and releasing operations. The purpose of those invocations is to show how the mentioned operations modify the partition table.

   The function shows the partitions and holes table contents this way:

```
[125] Main memory state (after allocating memory):
  [0] [0 -> 4][0 - ProgramName3]
  [1] [4 -> 12][HOLE]
  [2] [32 -> 96][1 - ProgramName6]
  [4] [128 -> 64][HOLE]
```

   First column shows the partition or hole number; second column, the features of the partition/hole (initial physical address and size); and, the third one, shows whether the partition is occupied (the PID of the process occupying the partition is displayed) or it is a hole.

   The text between parentheses is the parameter value for the function and must be one of the following two:

```
before allocating memory
after allocating memory
```

VERY IMPORTANT: main memory should be only allocated for processes successfully created and the corresponding previous messages only printed in that situation. If a process is not finally created, only the message displaying the amount of memory requested will be shown.

8. Add a function named `OperatingSystem_ReleaseMainMemory` that will be invoked during the finalization of a process. It must release the memory partition used by the process and show a message like the following (`SYSMEM` section, existing message number 45):

```
[123] Partition [0: 0-> 128] used by process [1 - Process1Name] has been released
```

In case this new hole is adjacent to other existing hole(s), it will be necessary to coalesce them into a single hole. Implement a function named `OperatingSystem_CoalesceHoles()` that examines whether it is necessary to coalesce holes and, if so, performs the appropriate operations and show this message (`SYSMEM` section, message number 144):

```
[72] Two_or_more_holes_has_been_coalesced
```

Besides, `OperatingSystem_ShowPartitionsAndHolesTable()` must be invoked before and after releasing the partition, using as a parameter the appropriate string between the following ones:

```
before releasing memory
after releasing memory
```

with a result like this:

```
[234] Main memory state (before releasing memory):
  [0] [0 -> 4][0 - ProgramNameInPartition0]
  …
```

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 … *

(**) 2*PROCESSTABLEMAXSIZE – 1

partitionsTable

initAddress size PID

0 1 2 … (**)

ReadyToRun (PID)

0 1 2 … (*)

USER

DAEM

Sleeping (PID)

0 1 2 … (*)

(*) PROCESSTABLEMAXSIZE – 1