Práctica 3: Gestión de Memoria

1º) Se nos pide que creemos en un lenguaje cualquiera un simulador de memoria en el que se pueda ver cómo funciona el algoritmo de siguiente hueco y el de peor hueco de asignación de memoria. Para ello nos dicen que la memoria máxima que habrá será de 2000 y que la interfaz que creemos tenga un menú en el que se puedan realizar, para cada opción, las funciones básicas de asignar y liberar memoria.

El ejercicio, y debido a que lo he visto más sencillo, he decidido programarlo en java, con un menú principal y varios submenús que muestran diferentes ejecuciones según lo que se seleccione.

Para poder ejecutar el programa se ha de realizar un “main” que ejecutar con el simulador java, en este caso, del entorno gráfico eclipse.

El código del main guardado en el archivo “Memoria.java” que está en el archivo comprimido que contiene esta memoria, es el siguiente:

**package** model;

/\*\*

\* **@author** ALEJANDRO REYES ALBILLAR 45931406-S

\* correo ara65@alu.ua.es

\*

\*/

**import** java.io.IOException;

/\*\*

\* The Class Memoria.

\*/

**public** **class** Memoria {

/\*\*

\* The main method.

\*

\* **@param** args the arguments

\* **@throws** NumberFormatException the number format exception

\* **@throws** IOException Signals that an I/O exception has occurred.

\*/

@SuppressWarnings("unused")

**public** **static** **void** main(String[] args) **throws** NumberFormatException, IOException {

Modulos modulos = **new** Modulos();

}

}

El código donde se programan todos los módulos, funciones y demás, llamado “Modulos.java” y que se encuentra junto a “Memoria.java”, es el siguiente:

**package** model;

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

/\*\*

\* The Class Modulos.

\*

\* **@author** ALEJANDRO REYES ALBILLAR 45931406-S

\* correo ara65@alu.ua.es

\*/

**public** **class** Modulos {

/\*\* Numero de procesos en lista. \*/

**int** nuevo = 0;// Solo es modificado a la hora de liberar memoria y asignar equivalentes cuando se introduce memoria en las diferentes funciones

/\*\* La posicion actual. \*/

**int** pos = 0;

/\*\* The br1. \*/

BufferedReader br1 = **new** BufferedReader(**new** InputStreamReader(System.***in***));

/\*\* The br2. \*/

BufferedReader br2 = **new** BufferedReader(**new** InputStreamReader(System.***in***));

/\*\* The br3. \*/

BufferedReader br3 = **new** BufferedReader(**new** InputStreamReader(System.***in***));

/\*\* The opcion. \*/

**private** **int** opcion = 0;

/\*\* The subopcion. \*/

**private** **int** subopcion = 0;

/\*\* The tam. \*/

**private** **int** tam = 0;

/\*\* The nom. \*/

**private** String nom;

/\*\* The max mem. \*/

**private** **int** MAX\_MEM = 2000;

/\*\* The equivalencias. \*/

**private** String[][] equivalencias = **new** String[MAX\_MEM][3];// 3 Columnas con MAX\_MEM filas posibles, almacenan [0]=nombre, [1]=sibolo, [2]=tamaño

/\*\* The Memoria. \*/

**private** String[] Memoria = **new** String[MAX\_MEM];// 50 filas 40 columnas=2000 bits de datos

/\*\*

\* Instantiates a new modulos.

\*

\* **@throws** NumberFormatException

\* the number format exception

\* **@throws** IOException

\* Signals that an I/O exception has occurred.

\*/

**public** Modulos() **throws** NumberFormatException, IOException {

inicializarMemoria();

System.***out***.println("Bienvenido al sistema de gestión de memoria con particiones dinámicas de la práctica 3 de Sistemas Operativos");

**try** {

**while** (opcion != 3) {

mainMenu();

opcion = Integer.*parseInt*(br1.readLine());

manageMainMenu(opcion);

}

} **catch** (java.lang.NumberFormatException nfe) {

System.***out***.println("No se pueden introducir dobles espacios ni caracteres extraños en la opcion, vuelva a intentarlo.\n");

@SuppressWarnings("unused")

Modulos modulos = **new** Modulos();

}

}

/\*\*

\* Devuelve la matriz en forma de string.

\*

\* **@return** devuelve un string

\*/

@Override

**public** String toString() {// Ya funciona

String s = "\n";

**int** pos = 0;

**for** (**int** i = 0; i < 50; i++) {// Filas

**for** (**int** j = 0; j < 40; j++) {// Columnas

**if** (j == 39) {

s = s + " | " + Memoria[pos] + " |\n";

pos++;

}

**else** **if** (j == 0) {

s = s + "| " + Memoria[pos];

pos++;

}

**else** {

s = s + " | " + Memoria[pos];

pos++;

}

}

}

**return** s;

}

/\*\*

\* Inicializar memoria.

\*/

**public** **void** inicializarMemoria() {// Funciona

**for** (**int** i = 0; i < MAX\_MEM; i++) {

Memoria[i] = " ";

}

}

/\*\*

\* Gets the aleat symbol.

\*

\* **@return** the aleat symbol

\*/

**private** String getAleatSymbol() {// Funciona

String rpta = (**char**) ((950 - 171 + 1) \* Math.*random*() + 171) + "";// A partir del 171 no nos encontraremos letras ni numeros en el código ascii

**return** rpta;

}

/\*\*

\* Liberar.

\* En caso de enontrar 2 nombres exatamente iguales solamente borrará el primero que encuentre siguiendo el orden de la lista

\*

\* **@param** nombre

\* the nombre

\*/

**public** **void** liberar(String nombre) {// Funciona

String delete = "";

**boolean** estar = **false**;// indica si está o no el valor en la memoria

**for** (**int** i = 0; i < MAX\_MEM; i++) {

**if** (nombre.equals(equivalencias[i][0]) && !estar) {

// Si encuentra un nombre igual y no lo habia encontrado antes entonces borrará.

// Esto hace que si hay nombres iguales borrará el primero que encuentre en la lista y se tendrá que repetir la operacion para borrar el resto.

delete = equivalencias[i][1];// El símbolo a borrar

equivalencias[i][0] = **null**;

equivalencias[i][1] = **null**;

equivalencias[i][2] = **null**;

estar = **true**;

organizarEquivalentes(i);

}

}

**for** (**int** j = 0; j < MAX\_MEM; j++) {

**if** (Memoria[j] == delete) {

Memoria[j] = " ";

**if** (j == MAX\_MEM - 1) {// Si libera la última posicion de memoria el puntero va al principio de la memoria

pos = 0;

System.***out***.println("He cambiado pos a 0.");

}

}

}

**if** (estar) {

System.***out***.println("Libero " + nombre);

imprimeEquivalencias();

System.***out***.println(toString());

}

**else** {

System.***out***.println("\nNo existe el proceso " + nombre + " en la lista.");

nuevo = 0;

}

}

/\*\*

\* Organizar equivalentes.

\*

\* **@param** pos

\* the pos

\*/

**public** **void** organizarEquivalentes(**int** pos) {// Funciona

**for** (**int** i = pos; i < MAX\_MEM - 1; i++) {

**if** (equivalencias[i + 1][0] != **null**) {

equivalencias[i][0] = equivalencias[i + 1][0];

equivalencias[i][1] = equivalencias[i + 1][1];

equivalencias[i][2] = equivalencias[i + 1][2];

equivalencias[i + 1][0] = **null**;

equivalencias[i + 1][1] = **null**;

equivalencias[i + 1][2] = **null**;

}

}

nuevo--;

}

/\*\*

\* Imprime equivalencias.

\*/

**public** **void** imprimeEquivalencias() {// Funciona

**if** (nuevo > 0) {

System.***out***.println("Nº | NOMBRE | SIMBOLO | TAMAÑO");

**for** (**int** i = 0; i < MAX\_MEM; i++) {

**if** (equivalencias[i][1] != **null**) {

System.***out***.println(i + " | " + equivalencias[i][0] + " | " + equivalencias[i][1] + " | " + equivalencias[i][2] + "\n");

}

}

}

}

/\*\*

\* asignarEquivalentes(String nombre, String simbolo, int tam): Asigna equivalentes

\*

\* **@param** nombre

\* the nombre

\* **@param** simbolo

\* the simbolo

\* **@param** tam

\* the tam

\*/

**public** **void** asignarEquivalentes(String nombre, String simbolo, **int** tam) {// Funciona

**if** (equivalencias[nuevo][0] == **null**) {

equivalencias[nuevo][0] = nombre;

equivalencias[nuevo][1] = simbolo;

equivalencias[nuevo][2] = String.*valueOf*(tam);

nuevo++;

}

imprimeEquivalencias();

}

/\*\*

\* primeraLibre(int pos): Primera libre desde la posición actual del puntero.

\*

\* **@param** pos

\* the pos

\* **@return** the int

\*/

**public** **int** primeraLibre(**int** pos) {// Funciona

**int** count = 0;

**for** (**int** i = pos; i < MAX\_MEM; i++) {

**if** (Memoria[i] == " " && count == 0) {

pos = i;

count++;

}

**else** **if** (i == MAX\_MEM - 1 && count == 0) {

pos = 0;

}

}

System.***out***.println("La primera libre es " + pos);

**return** pos;

}

/\*\*

\* isFree(int tam, int pos): indica si hay un hueco para colocar el proceso completo dentro de la memoria

\*

\* **@param** tam

\* the tam

\* **@param** pos

\* the pos

\* **@return** true, if is free

\*/

**public** **boolean** isFree(**int** tam, **int** pos) {// Funciona

// Hay un hueco para colocar el proceso completo dentro de la memoria

**boolean** free = **true**;

**if** (pos != MAX\_MEM && (MAX\_MEM - (pos + 1)) > tam) {// Si no estamos en el final de la memoria y existe espacio en memoria

**for** (**int** i = 0; i < tam; i++) {// Se comprueba que el tamaño cabe en la memoria

**if** (Memoria[pos + i] != " ") {

free = **false**;

}

}

}

**else** {

pos = 0;

pos = primeraLibre(pos);

System.***out***.println(pos);

}

**return** free;

}

/\*\*

\* Colocar.

\*

\* **@param** pos

\* the pos

\* **@param** tam

\* the tam

\* **@param** simbolo

\* the simbolo

\*/

**public** **void** colocar(**int** pos, **int** tam, String simbolo) {// Funciona

// Solo se le llama cuando puede meter fila.

**for** (**int** i = 0; i < tam; i++) {// Filas

Memoria[pos + i] = simbolo;

}

}

/\*\*

\* quedahueco(): Devuelve un int con el tamaño que queda libre en memoria

\*

\* **@return** the int

\*/

**public** **int** quedahueco() {// Funciona

**int** queda = MAX\_MEM;

**for** (**int** i = 0; i < MAX\_MEM; i++) {

**if** (equivalencias[i][0] != **null**) {

queda = queda - Integer.*parseInt*(equivalencias[i][2]);

}

}

**return** queda;

}

/\*\*

\* siguienteHueco(String nombre, int tam): Asigna un valor a memoria mediante el algoritmo de Siguiente hueco.

\*

\* **@param** nombre

\* the nombre

\* **@param** tam

\* the tam

\*/

**public** **void** siguienteHueco(String nombre, **int** tam) {// Funciona

**if** (tam > 0) {

**if** (quedahueco() > 0) {

**if** (pos != MAX\_MEM) {

String simbolo = getAleatSymbol();

**if** (isFree(tam, pos)) {

System.***out***.println("Utilizo siguiente hueco para guardar " + nombre + " de tamaño " + tam);

// introducir a tabla de equivalencias

asignarEquivalentes(nombre, simbolo, tam);

pos = primeraLibre(pos);

colocar(pos, tam, simbolo);

pos = primeraLibre(pos);

}

**else** {

**while** (!isFree(tam, pos)) {

pos = primeraLibre(pos);

}

System.***out***.println("Utilizo siguiente hueco para guardar " + nombre + " de tamaño " + tam);

// introducir a tabla de equivalencias

asignarEquivalentes(nombre, simbolo, tam);

colocar(pos, tam, simbolo);

pos = primeraLibre(pos);

}

System.***out***.println(toString());

System.***out***.println("Queda " + quedahueco() + " espacio libre en memoria.");

}

**else** {

System.***out***.println("\nNo queda más espacio en la memoria, libere procesos.");

}

}

**else** {

System.***out***.println("No queda memoria suficiente para introducir el proceso que usted desea.");

}

}

**else** {

System.***out***.println("\nEl proceso que ha intentado introducir no consume memoria, por tanto será ignorado.");

}

}

/\*\*

\* maximoHueco(): Busca el hueco máximo entre todos los que hay en la matriz

\*

\* **@param** tam

\* es el tamaño de memoria que queremos reservar

\* **@return** devuelve la posición del hueco más grande de la memoria

\*/

**public** **int** maximoHueco(**int** tam) {

**int** max = 0;

**int** pos = primeraLibre(0);// Aqui tenemos, del primer hueco libre, su posición

**int** aux = 0;

**int** count = 0;

**if** (quedahueco() >= tam) {// Si queda hueco suficiente en la matriz para introducir el proceso

**for** (**int** i = pos; i < Memoria.length; i++) {

**if** (Memoria[i] == " " && count == 0) {// Si encuentra un primer hueco

max++;

}

**else** **if** (Memoria[i] != " ") {// Si termina cualquier hueco

count = 1;

aux = 0;

}

**else** **if** (Memoria[i] == " " && count == 1) {// Si encuentra un segundo, tercer, cuarto, ... hueco

aux++;

**if** (aux > max) {

max = aux;

pos = i - max + 1;

}

}

}

**return** pos;

}

**else** {

System.***out***.println("No queda espacio en memoria para introducir el proceso seleccionado.");

**return** -1;

}

}

/\*\*

\* peorHueco(String nombre int tam): Asigna un valor a memoria mediante el algoritmo de Siguiente hueco.

\*

\* **@param** nombre

\* the nombre

\* **@param** tam

\* the tam

\*/

**public** **void** peorHueco(String nombre, **int** tam) {

String simbolo = getAleatSymbol();

**int** pos = maximoHueco(tam);

**if** (pos > -1) {

**if** (isFree(tam, pos)) {

System.***out***.println("Utilizo peor hueco para guardar " + nombre + " de tamaño " + tam + "\n");

asignarEquivalentes(nombre, simbolo, tam);

colocar(pos, tam, simbolo);

System.***out***.println(toString());

System.***out***.println("Queda " + quedahueco() + " espacio libre en memoria.");

}

**else** {

System.***out***.println("No se ha podido asignar correctamente la memoria con este algoritmo, pruebe a liberar memoria o intentelo con otro algoritmo.\n");

}

}

**else** {

System.***out***.println("No se ha podido asignar correctamente la memoria con este algoritmo, pruebe a liberar memoria o intentelo con otro algoritmo.\n");

}

}

/\*\*

\* Main menu.

\*/

**public** **void** mainMenu() {

System.***out***.println("\nSeleccione el aloritmo que desea utilizar para la gestión de memoria:\n"

+ "1. Algoritmo de Siguiente Hueco.\n"

+ "2. Algoritmo de Peor Hueco.\n"

+ "3. Salir.");

}

/\*\*

\* Menu1.

\*/

**public** **void** menu1() {

System.***out***.println("\nHa seleccionado Algoritmo de Siguiente Hueco\n"

+ "Seleccione la acción que quiere llevar a cabo:\n"

+ "1.Asignar Memoria.\n"

+ "2.Liberar Memoria.\n"

+ "3.Menu anterior.");

}

/\*\*

\* Menu2.

\*/

**public** **void** menu2() {

System.***out***.println("\nHa seleccionado Algoritmo de Peor Hueco\n"

+ "Seleccione la acción que quiere llevar a cabo:\n"

+ "1.Asignar Memoria.\n"

+ "2.Liberar Memoria.\n"

+ "3.Menu anterior.");

}

/\*\*

\* Leer datos.

\*

\* **@throws** IOException

\* Signals that an I/O exception has occurred.

\*/

**public** **void** leerDatos() **throws** IOException {

System.***out***.println("\nIntroduzca el nombre del proceso:");

nom = br2.readLine();

**try** {

System.***out***.println("\nIntroduzca el tamaño del proceso:");

tam = Integer.*parseInt*(br3.readLine());

} **catch** (java.lang.NumberFormatException nfe) {

System.***out***.println("No se ha introducido un tamaño de proceso. Vuelva a intentarllo");

leerDatos();

}

**while** (tam > MAX\_MEM || tam > quedahueco()) {

System.***out***.println("\nLa memoria es de tamaño " + MAX\_MEM + " por lo que no puedes introducir un valor mayor a ese.\nPor favor, inténtalo de nuevo.");

System.***out***.println("\nIntroduzca el tamaño del proceso:");

tam = Integer.*parseInt*(br3.readLine());

}

}

/\*\*

\* Leer liberar.

\*

\* **@throws** IOException

\* Signals that an I/O exception has occurred.

\*/

**public** **void** leerLiberar() **throws** IOException {

System.***out***.println("\nIntroduzca el nombre del proceso a liberar:");

nom = br2.readLine();

}

/\*\*

\* Manage sub menu1.

\*

\* **@param** subopcion

\* the subopcion

\* **@throws** NumberFormatException

\* the number format exception

\* **@throws** IOException

\* Signals that an I/O exception has occurred.

\*/

**public** **void** manageSubMenu1(**int** subopcion) **throws** NumberFormatException, IOException {

**while** (subopcion != 3) {

menu1();

subopcion = Integer.*parseInt*(br3.readLine());

**switch** (subopcion) {

**case** 1:

leerDatos();

siguienteHueco(nom, tam);

subopcion = 0;

**break**;

**case** 2:

leerLiberar();

liberar(nom);

subopcion = 0;

**break**;

**case** 3:

opcion = 0;

**break**;

**default**:

System.***out***.println("Has seleccionado una opcion incorrecta");

}

}

}

/\*\*

\* Manage sub menu2.

\*

\* **@param** subopcion

\* the subopcion

\* **@throws** IOException

\* Signals that an I/O exception has occurred.

\*/

**public** **void** manageSubMenu2(**int** subopcion) **throws** IOException {

**while** (subopcion != 3) {

menu2();

subopcion = Integer.*parseInt*(br3.readLine());

**switch** (subopcion) {

**case** 1:

leerDatos();

peorHueco(nom, tam);

subopcion = 0;

**break**;

**case** 2:

leerLiberar();

liberar(nom);

subopcion = 0;

**break**;

**case** 3:

opcion = 0;

**break**;

**default**:

System.***out***.println("Has seleccionado una opcion incorrecta");

}

}

}

/\*\*

\* Manage main menu.

\*

\* **@param** opcion

\* the opcion

\* **@throws** NumberFormatException

\* the number format exception

\* **@throws** IOException

\* Signals that an I/O exception has occurred.

\*/

**public** **void** manageMainMenu(**int** opcion) **throws** NumberFormatException, IOException {

**try** {

**switch** (opcion) {

**case** 1:

manageSubMenu1(subopcion);

**break**;

**case** 2:

manageSubMenu2(subopcion);

**break**;

**case** 3:

System.***out***.println("¡¡Hasta Pronto!!");

**break**;

**default**:

System.***out***.println("Has seleccionado una opción incorrecta.\n");

**break**;

}

} **catch** (java.lang.NumberFormatException nfe) {

System.***out***.println("No se pueden introducir dobles espacios ni caracteres extraños en la opcion, vuelva a intentarlo.");

manageMainMenu(opcion);

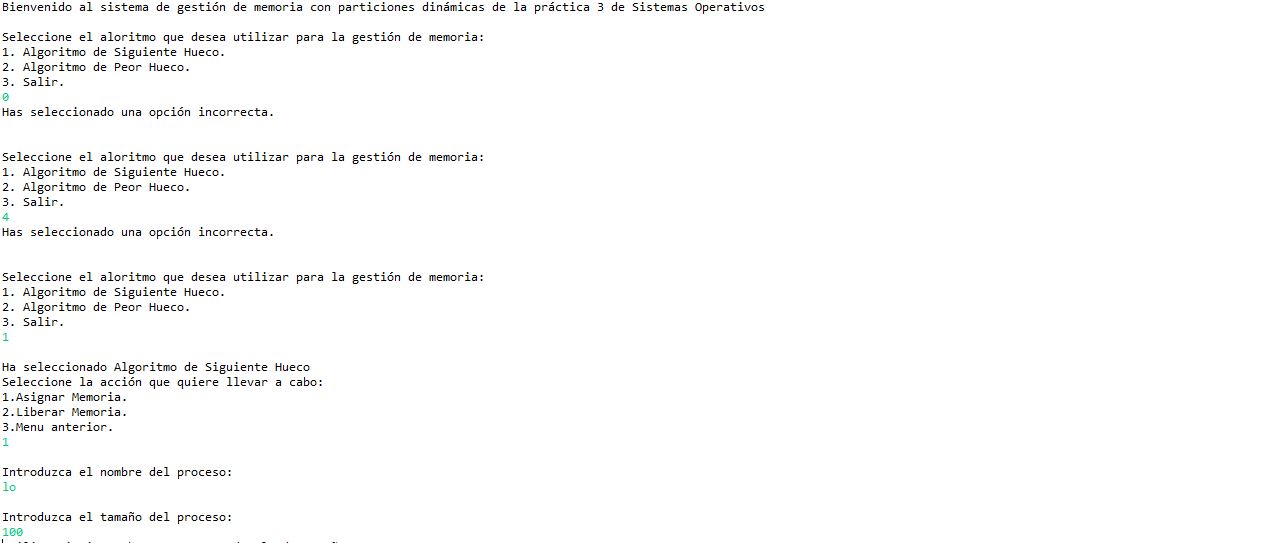
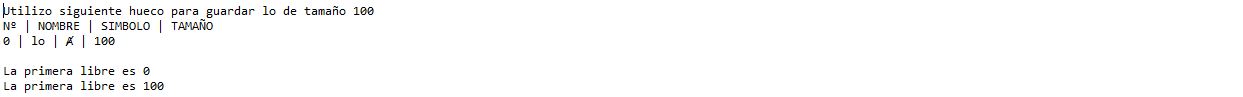
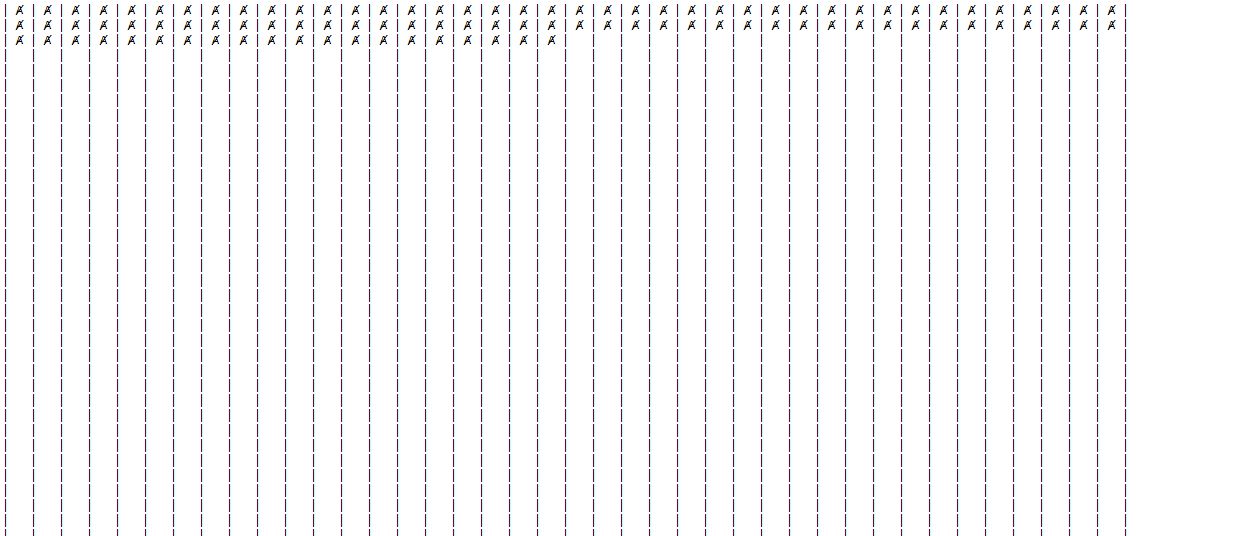
}

}

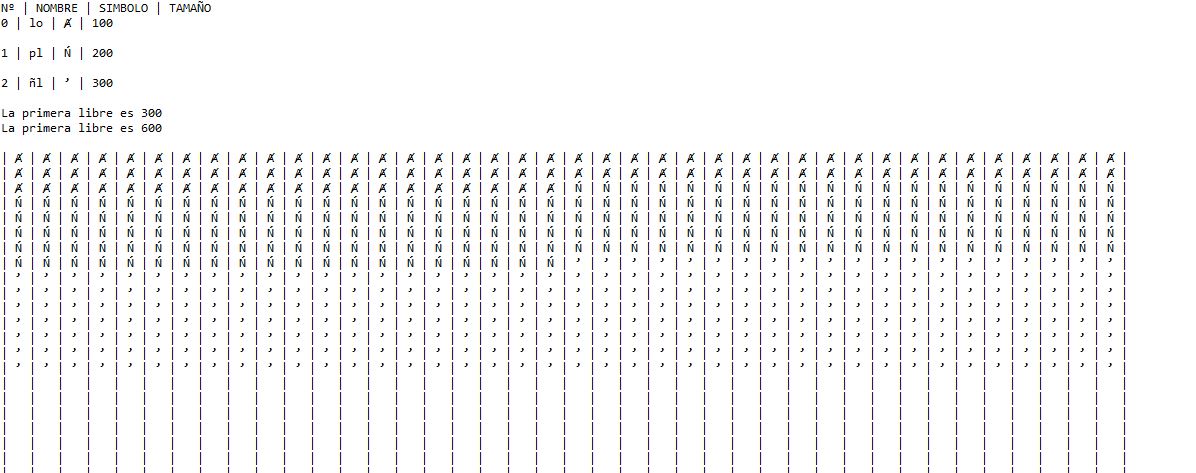
}

En este archivo se manejan diferentes módulos que controlan que se introduzca una opción correcta, que una posición de memoria está ocupada o no, cual es la primera posición de memoria ocupada y los dos algoritmos de siguiente y peor hueco además del usado para liberar memoria. El resultado se mostrará por tablero de comandos con una matriz de 50 de alto por 40 de ancho. Se utiliza un Array de tamaño 2000 que se recorre y en el que se introduce un carácter aleatorio creado por una función interna. En el caso de que dos funciones se llamen igual se limpiará la primera de las dos que se haya introducido en la lista de Array bidimensional de altura 2000 y anchura 3.

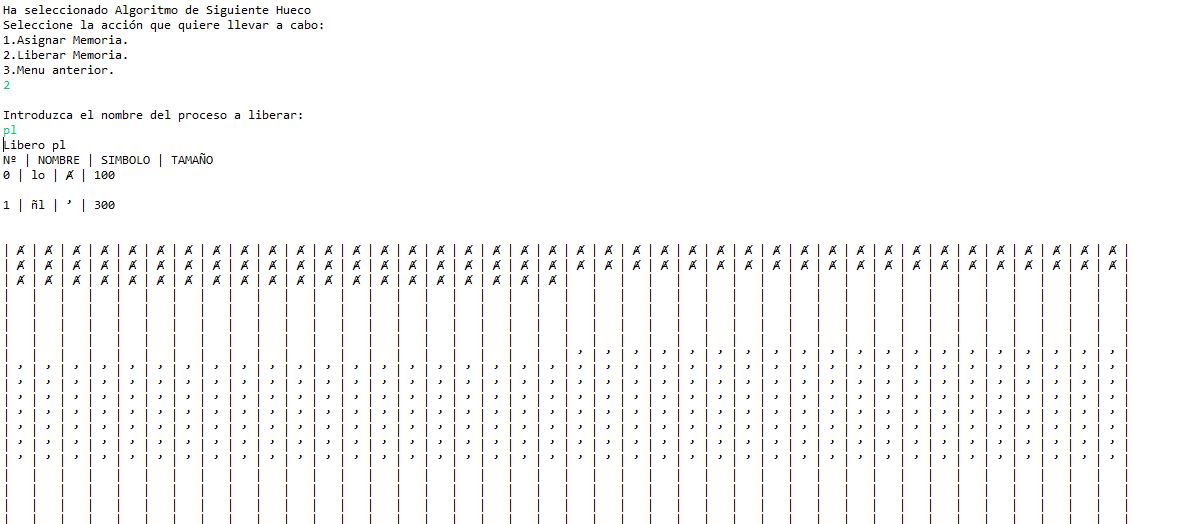
Aquí se muestran algunas capturas de la ejecución:



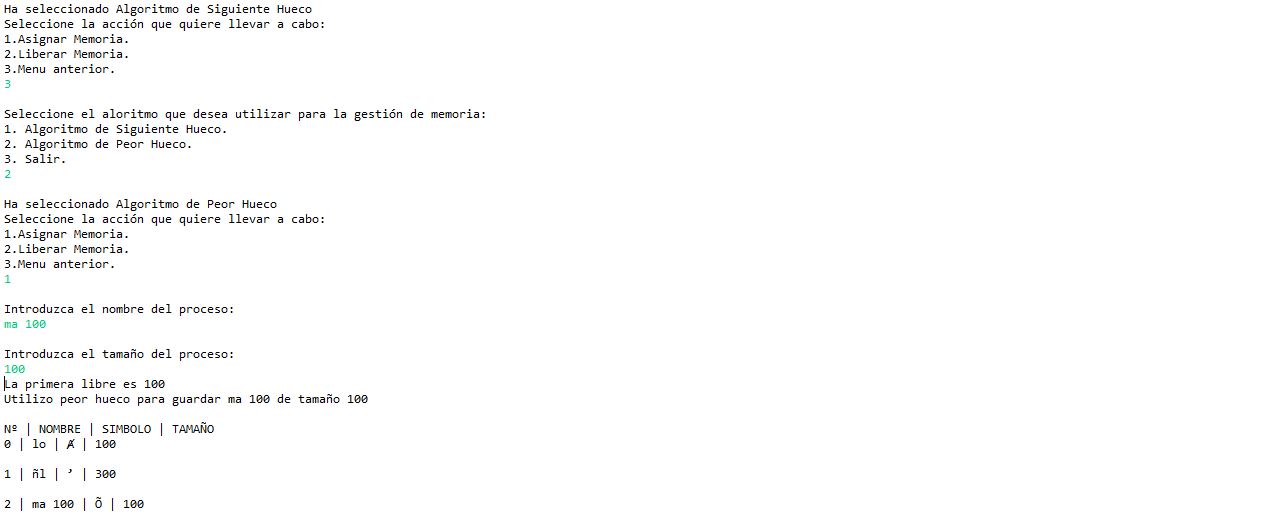
Tras unas cuantas asignaciones:



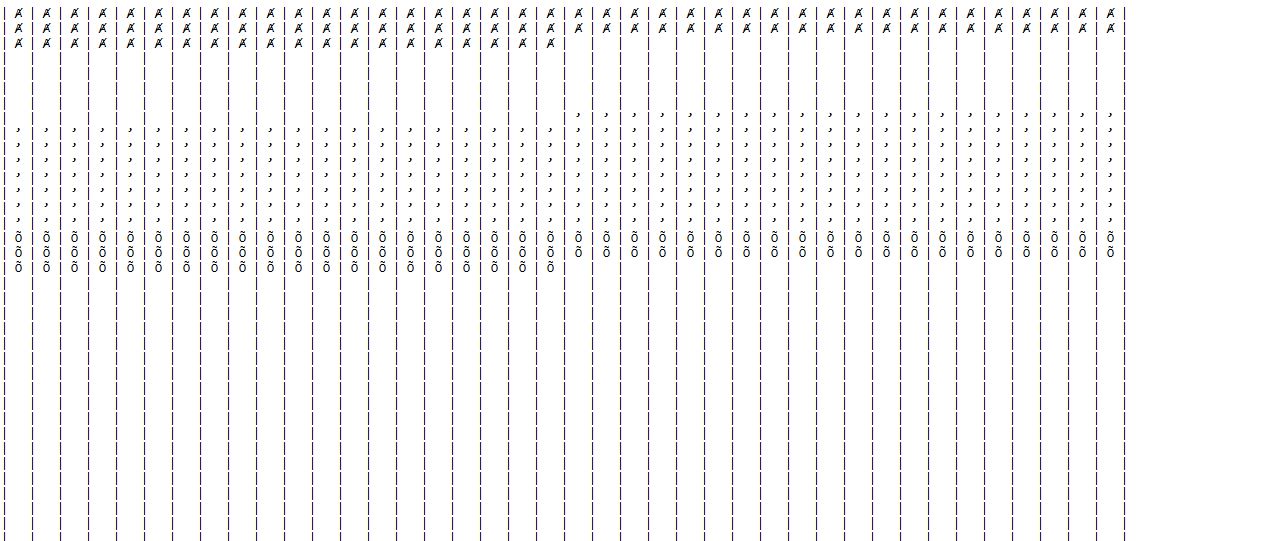
Ahora liberamos un proceso, el “pl” que es el intermedio:



A continuación cambiamos de algoritmo al del peor hueco:



Podemos observar que el algoritmo de peor hueco busca en la matriz el hueco de memoria más grande e introduce el proceso, si cabe, en el hueco en cuestión.



Se han implementado otras opciones como errores por memoria llena y otras tantas que se pueden descubrir realizando diferentes ejecuciones con el código que también se adjunta.

