Sistemas Operativos: Práctica 3

Hecha por:
Aaron Blasco Tremiño (74379762Q) (abtb2@alu.ua.es)
Rubén Gómez Baró (29523336F) (rgbg1@alu.ua.es)

Introducción

Hemos elegido javascript como lenguaje para implementar esta práctica. De esta manera, para mostrarlo gráficamente hemos diseñado una página web local con html y css. La mayor parte del código de la práctica se encuentra en el html, css y javascript encargados de la presentación. Sin embargo, profundizaremos únicamente en el código que más concierne a esta práctica, es decir, aquel que se encarga de la gestión de las particiones (todas las funciones se encuentran en main.js).

Por otro lado, consideramos importante mencionar que el programa ha sido desarrollado y probado en windows por el navegador chrome v. 108.0.5359.125 (official build) (64 bits). El programa da por hecho que el texto del archivo de entrada tiene un formato correcto y, por tanto, no garantizamos que funcione bien en caso contrario. Por último, para una visualización más cómoda se eliminan los instantes duplicados, es decir, aquel instante que no se muestre és porque es igual al anterior y se ha obviado. Aclarado esto, comencemos.

Variables y constantes globales

```
const MINIMUM_MEMORY_REQUIRED = 100; // Minimum memory that a process m
const MEMORY_CAPACITY = 2000;
var initialInstant = 1;              // First instant shown
const processes = [];                // Array that contains all the val
const instants = [];                 // Contains all instants except fo
const lists = [];                    // contains a ul element for each

const lists_container = document.getElementById("lists_container"); //
```

NOTA: Durante toda la memoria, en las capturas de pantalla vamos a cortar los comentarios para que el código principal se vea más grande

Como constantes tenemos:

- **MINIMUM_MEMORY_REQUIRED: la unidad de asignación mínima y que se utilizará como número de corte para ignorar procesos con menor memoria requerida**
- **MEMORY_CAPACITY: el nombre es bastante autoexplicativo, indica la capacidad de la memoria**

También tenemos las siguientes variables globales:

- **initialInstant: primer instante que se muestra, cabe recalcar que se utilizará como índice, por tanto puede que el primer instante que se muestre tenga un identificador diferente**
- **processes: array que contiene los procesos leídos del archivo de texto**
- **instants: array que contiene todos los instantes, eliminados los duplicados**
- **lists: contiene elementos ul, que son los que se muestran en la página**
-

Por último, como elemento HTML tenemos lists_container: div donde se anexarán las listas de lists

Clases

Process

```
class Process {
  constructor(name, arrivalInstant, memoryRequired, executionTime) {
    this.name = name;
    this.arrivalInstant = parseInt(arrivalInstant);
    this.memoryRequired = parseInt(memoryRequired);
    this.departureInstant = parseInt(executionTime) + this.arrivalInstant;
  }
  advanceInstant() {
    this.departureInstant--;
    this.arrivalInstant--;
  }
  delay() {
    this.departureInstant++;
    this.arrivalInstant++;
  }
  setstartAddress(startAddress) {
    this.startAddress = startAddress;
    this.endAddress = startAddress + this.memoryRequired - 1;
  }
  getSize() { // I only wrote this function so that a Process instance can
    return this.memoryRequired;
  }
}
```

NOTA: parseInt transforma de string a int

Esta clase representa un proceso, se construye con los datos que se nos ofrecen en el archivo en el mismo orden que en el enunciado. Sin embargo, en lugar de guardar el tiempo de ejecución, se guarda el instante en el que el proceso terminará y, por tanto, se irá de la memoria. Además hemos añadido un método para que los instantes de llegada y de partida decrezcan en uno (cuando se avance al siguiente instante) y otro para que crezcan en uno también (para cuando no haya espacio en la memoria y tenga que esperar al siguiente instante). Por otro lado, cuenta con un método para establecer su dirección en la memoria y un getter para la memoria requerida. Tal y como pone en el comentario, la única razón para añadir el getter fue poder tratar igual tanto a un proceso como a una partición vacía (siguiente clase) cuando se quiera comparar tamaños.

Partición vacía (EmptyPartition)

```
class EmptyPartition {
  constructor(startAddress, endAddress) {
    this.startAddress = startAddress;
    this.endAddress = endAddress;
  }
  getSize() {
    return this.endAddress - this.startAddress + 1;
  }
}
```

Esta clase únicamente cuenta con un constructor, que requiere de sus direcciones de inicio y de fin, y un método que devuelve su tamaño. El más uno se debe a que la dirección de inicio también cuenta, de manera que una partición vacía que ocupe desde la dirección de memoria 0 hasta la 99, estaría ocupando 100 espacios en lugar de 99.

ProcessFile(file)

```
function processFile(file) {
  // First of all, the selected file is readed by a FileReader
  const reader = new FileReader();

  reader.readAsText(file);
  reader.addEventListener("load", () => {
    // Transform the string into the processes we will work with
    interpretResult(reader.result, processes);

    // The instant 0 shows the empty state of the memory before any process could arrive. In addi
    let instant = 0;

    instants.push({instant: instant, memory: [new EmptyPartition(0, MEMORY_CAPACITY - 1)]}) // Ev
                                                                                            // ar

    // The loop will create new instants as long as there's at least one process that hasn't fini
    while (processes.some((process) => process.departureInstant > 0)) {
      instant++;
      processes.map(process => process.advanceInstant()) // little reminder: advanceInstant decre
      newInstant(instant);                               //                  so processes arrive
    }

    // Remove duplicates
    for (let i = 0; i < instants.length - 1; i++)
      if (instants[i].memory.equals(instants[i+1].memory)) {
        instants.splice(i + 1, 1);
        i--;
      }
```

```
    // If the initial instant doesn't exist, it starts with the 0 on
    if (!instants[initialInstant]) {
      console.log("hola");
      initialInstant = 0;
      // As there won't be a previous instant, the "previous" button
      disable(document.getElementById("previous"));
      // If besides of not existing the initial instant, there's jus
      if (instants.length == 1)
        disable(document.getElementById("next"));
    }

    // Transforms the instants into showable litsts
    createLists();

    // Firstly, the display "none" is changed
    lists[initialInstant].style.display = "initial";

    // Then, the first list is shown
    setTimeout(() => {
      lists[initialInstant].classList.add("show");
      lists[initialInstant].style.transform = "skewY(-15deg)";
    }, 300);
  });
}
```

NOTA: Se me ha colado el log("hola"), pese a que en el código lo he borrado he decidido dejarlo en la captura porque me ha hecho gracia

Como parámetro tenemos el archivo de entrada, con el cual trabajaremos. Para empezar declaramos un FileReader con el cual leemos el archivo. Una vez leído damos paso a la función interpretResult(content, processes), que se encarga de transformar el texto en forma de string a los procesos con los que trabajaremos (se almacenan en processes).

Empezamos a por añadir el instante 0. Hemos diseñado los instantes como un objeto que tiene un identificador: instant, y un array con particiones vacías y procesos: memory. De esta manera, el instante 0 es un objeto con identificador igual a 0 y cuya memoria únicamente contiene una partición vacía que se extiende desde la dirección 0 hasta la última dirección de la memoria (1999 en este caso). Continuamos con la construcción de instants con un bucle while que se ejecutará mientras quede al menos un instante por abandonar la memoria. En cada iteración del bucle se avanza un instante, tanto sumando uno a la variable instant (variable que servirá de identificador para cada nuevo instante) como ejecutando el método advanceInstant() de cada uno de los instantes, y se añade un nuevo instante mediante la función newInstant(instant). Para concluir la construcción de instants procedemos a eliminar los duplicados mediante un simple bucle for. Cabe puntualizar que el instante que se conserva es el primero de todos los duplicados, es decir, si el instante 3, 4 y 5 son iguales, aquel que se conserva es el 3.

La última parte de la función está dedicada a la presentación de los instantes. Primero se comprueba que el instante que se planea mostrar primero exista y se toman medidas en caso contrario. Posteriormente se crean los elementos HTML de tipo ul mediante la función createLists(). Por último se muestra la primera lista de todas y se continúa con el progreso de la página

interpretResult(content, processes)

```
// Gets the processes from the string read from the file
function interpretResult(content, processes) {
  content = content.replace("\r", "");     // As far as i'm concerned, \n\r is used in windows as a new li
  const lines = content.split("\n");       // lines: the content separated in lines
  const words = [];                        // words: every line is also separated in words. it is a two-di

  for (let line of lines) {
    words.push(line.split(" "));
  }

  for (let line of words) {              /* words is a two-dimesional array, this means every words[x][x]
                                            However, every words[x] is an array of words that represents a
                                            That's why the parameter is called line */

    // little reminder:
    // line[0] = name of the process
    // line[1] = arrivalInstant, must be positive a positive value
    // line[2] = memoryRequired, must not exceed memory capacity nor be lower than MINIMUM_MEMORY_REQUIRE
    // line[3] = executionTime, must be positive a positive value
    if (line[1] > 0 && line[2] <= MEMORY_CAPACITY && line[2] >= MINIMUM_MEMORY_REQUIRED && line[3] > 0)
      processes.push(new Process(line[0], line[1], line[2], line[3]));
  }
}
```

Primero, eliminamos los posibles '\r' del string. Por lo que tenemos entendido, es un carácter que se usa en windows acompañado del universal '\n' para indicar un salto de línea. No obstante, no se utiliza en linux ni en mac desde hace mucho tiempo.
Después separamos cada línea del string y posteriormente cada una en palabras, de manera que nos queda un array bidimensional en el que cada words[x] expresa una línea y cada words[x][y] una palabra. Por último, por cada words[x], añadimos un proceso a processes si y solo si pasa las siguientes comprobaciones:

- **Tanto el instante de llegada como el tiempo de ejecución han de ser mayores que cero**
- **La memoria requerida debe ser mínimo de 100 y no puede sobrepasar la capacidad de la memoria**

newInstant(instant)

```
// Creates a new Instant
function newInstant(instant) {
  // The memory of the previous instant is used as a start
  let memory = [...instants[instant - 1].memory];

  eraseProcesses(memory);

  addProcesses(memory);

  instants.push({instant: instant, memory: memory})
}
```

En esta función no hay mucho que explicar, los pasos para crear un nuevo instante son sencillos. Utilizamos como punto de partida la memoria del instante anterior. Primero se eliminan los procesos que han finalizado (eraseProcesses(memory)) y luego añadimos los que deben llegar en el instante en cuestión (addProcesses(memory)). Por último, solo queda añadir el nuevo instante a instants

eraseProcesses(memory)

```
function eraseProcesses(memory) {
  // Starts by looking for the processes that have finished
  const finalizedProcesses = [];
  for (let process of processes)
    if (process.departureInstant == 0)
      finalizedProcesses.push(process);

  // Replaces each finalized process for an empty partition with placed at the same spot
  for (let process of finalizedProcesses)
    memory.splice(memory.indexOf(process), 1, new EmptyPartition(process.startAddress,process.endAddress));

  compactEmptyPartitions(memory);
}
```

Comenzamos por identificar los procesos que abandonan la memoria en este instante, aquellos cuyo instante de partida ha llegado a 0. Recordemos que antes de crear cada instante se reduce en uno el tiempo de llegada y el de partida de todos los instantes (ProcessFile(file)).

Una vez identificados, sustituimos cada uno por una partición vacía con la misma dirección de inicio y de fin. Para terminar unimos en una todas aquellas particiones vacías que se encuentren seguidas en la memoria mediante la función compactEmptyPartitions(memory).

compactEmptyPartitions(memory)

```
// Compats the small empty partitions in a row into a big one
function compactEmptyPartitions(memory) {
  for (let i = 0; i < memory.length - 1; i++)
    // If after the current empty partition there's another, both are replaced for a big one
    if (memory[i] instanceof EmptyPartition && memory[i + 1] instanceof EmptyPartition) {
      memory.splice(i, 2, new EmptyPartition(memory[i].startAddress, memory[i + 1].endAddress));
      i--;
    }
}
```

Por cada partición de la memoria, si se trata de una vacía y la siguiente también, ambas se sustituyen por una sola con la dirección de inicio de la primera partición y la dirección de fin de la segunda

addProcesses(memory)

```
// Adds a process to the memory
function addProcesses(memory) {
  // Starts by looking for the empty partitions, as well as the processes that should arrive
  const emptyPartitions = [];
  for (partition of memory)
    if (partition instanceof EmptyPartition)
      emptyPartitions.push(partition);

  const arrivalProcesses = [];
    for (let process of processes)
      if (process.arrivalInstant == 0)
        arrivalProcesses.push(process);

  // For each process that should arrive
  for (let process of arrivalProcesses) {
    // If there's an empty partition equal or greater than the memory that requires
    if (hasEmptySpace(emptyPartitions, process.memoryRequired)) {
      // Sets the process start address in the best empty partition depending on the mode sel
      process.setstartAddress(bestEmptyPartition(emptyPartitions, process.memoryRequired));
      // Cuts the empty partition placed in the address to make room for the process
      cutEmptySpace(process.startAddress, process.memoryRequired, emptyPartitions);
    } else
      // If there's no room for the process in memory, the process will try again in the next
      process.delay();
  }

  // Erases the empty spaces in the initial memory to later push the new empty spaces
  for (let i = 0; i < memory.length; i++)
    if (memory[i] instanceof EmptyPartition) {
      memory.splice(i, 1);
      i--;
    }
```

```
    }

  for (let i = 0; i < emptyPartitions.length; i++)
    memory.push(emptyPartitions[i]);

  // The processes that haven't been delayed are pu
  for (let i = 0; i < arrivalProcesses.length; i++)
    if (arrivalProcesses[i].arrivalInstant == 0)
      memory.push(arrivalProcesses[i]);

  // Orders the partitions that have been pushed in
  memory.sort(comparePartitions);
}
```

Identificamos tanto las particiones vacías (emptyPartitions), como los procesos que deberían llegar (arrivalProcesses).

Por cada uno de los procesos de arrivalProcesses: si queda suficiente espacio en la memoria (lo cual se comprueba con la función hasEmptySpace(emptyPartitions, memoryRequired)) se busca el mejor espacio dependiendo de si el usuario ha elegido mejor hueco o peor hueco mediante la función bestEmptyPartition(emptyPartitions, memoryRequired) y se asigna su dirección de inicio al proceso. Acto seguido, se desplaza la dirección de inicio de la partición vacía elegida mediante la función cutEmptySpace(startAddress, size, emptyPartitions). En caso de que no haya espacio para el proceso, se llama a su método de instancia delay() (Process) de manera que el proceso volverá a intentar entrar en la memoria en el siguiente instante.

Para terminar se forma la memoria con las nuevas particiones vacías y los nuevos procesos: se borran las particiones vacías que había en la memoria y se añaden las del array emptyPartitions, array con el que se ha estado trabajando. Justo después se añaden a la memoria los procesos que no se han retrasado un instante y, finalmente, se ordenan las particiones de la memoria en función de su dirección de inicio mediante la función Array.prototype.sort(), que a su vez llama a la función compare Partitions(a, b).

hasEmptySpace(emptyPartitions, memoryRequired)

```
// Checks if there's room for a process in memory
function hasEmptySpace(emptyPartitions, memoryRequired) {
  for (let partition of emptyPartitions)
    if (partition.getSize() >= memoryRequired)
      return true;
  return false;
}
```

Por cada partición, si su tamaño es mayor o igual al requerido se devuelve true. En caso contrario se devuelve false

bestEmptyPartition(emptyPartitions, memoryRequired)

```
// Finds the best empty partition depending on the mode selected
function bestEmptyPartition(emptyPartitions, memoryRequired) {
  let best;
  if (getMode()) { // little reminder: true: best gap; false: worst gap
    // As the objective is to find the smallest one, the biggest possible empty partition is
    best = new EmptyPartition(0, MEMORY_CAPACITY - 1);
    for (let partition of emptyPartitions)
      // If the partition is big enough for the process to fit in it and it's smaller than t
      if (partition.getSize() >= memoryRequired && partition.getSize() < best.getSize())
        best = partition;
  } else {
    // It works the same way as in the best gap case, but looking for the largest empty part
    best = new EmptyPartition(0, 0);
    for (let partition of emptyPartitions)
      if (partition.getSize() >= memoryRequired && partition.getSize() > best.getSize())
        best = partition;
  }
  // Finally returns the start address of the best partition found
  return best.startAddress;
}
```

Esta función se divide en dos dependiendo del modo que el usuario haya elegido. Sin embargo, ambos funcionan de manera muy similar. En ambos casos se define una partición, la peor posible. Para mejor hueco la más grande que cabe en la memoria y para peor hueco la más pequeña, ocupando una sola unidad. A partir de ahí se recorren las particiones vacías para encontrar la más pequeña/grande que además sea mayor o igual que la memoria requerida. Se finaliza devolviendo la dirección de inicio de la mejor partición encontrada.

cutEmptySpace(startAddress, size, emptyPartitions)

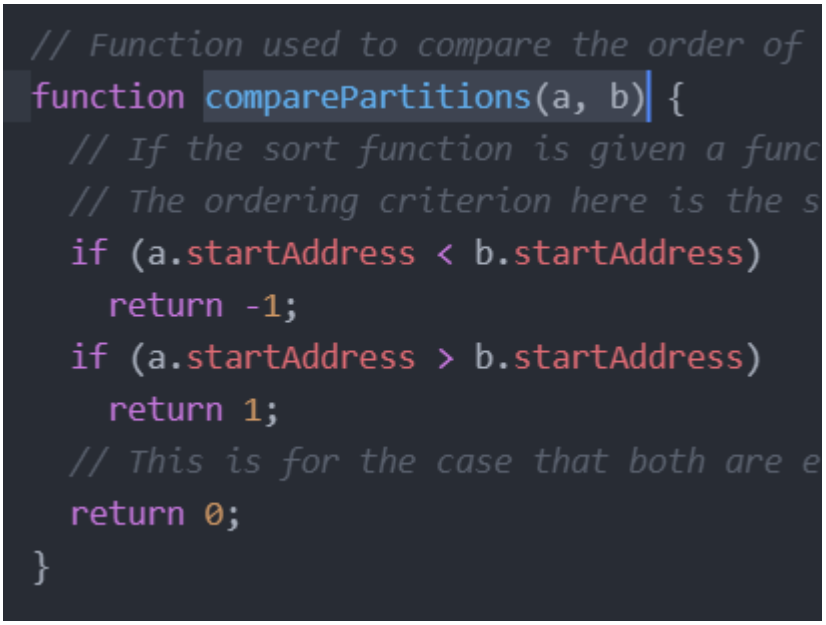
```
// Resizes an empty space for a process to fit in the gap left
function cutEmptySpace(startAddress, size, emptyPartitions) {
  let i;
  // First of all, looks for the empty partition to cut
  for (i in emptyPartitions)
    if (emptyPartitions[i].startAddress == startAddress)
      break;

  // As the index isn't declared in the for, once it breaks out of it, the index is still accesible
  // The empty partition is replaced for one whose start address is moved after the size of the process that's gonna
  emptyPartitions[i] = new EmptyPartition(emptyPartitions[i].startAddress + size, emptyPartitions[i].endAddress);

  // In the case that the empty partition had the same size as the required (so now has null size), it's erased
  if (emptyPartitions[i].getSize() <= 0) // The case where the empty partition finishes with a negative size is check
    emptyPartitions.splice(i, 1);
}
```

Se busca entre las particiones vacías aquella cuya dirección de inicio se desea desplazar. Una vez la encontramos, la sustituimos por una partición vacía nueva con la dirección de inicio anterior más la memoria requerida del proceso que la desplaza. Por último comprobamos si la partición tenía el mismo tamaño que el proceso y por tanto el nuevo tamaño de la función es 0 o menor, en cuyo caso la eliminamos. Nótese que para buscar la partición hemos definido el índice fuera del for, de manera que cuando se encuentra la partición, el bucle se corta y el índice nos sirve para trabajar con la partición fuera del for

comparePartitions(a, b)



```
// Function used to compare the order of
function comparePartitions(a, b) {
  // If the sort function is given a func
  // The ordering criterion here is the s
  if (a.startAddress < b.startAddress)
    return -1;
  if (a.startAddress > b.startAddress)
    return 1;
  // This is for the case that both are e
  return 0;
}
```

Las funciones que se usan como parámetro para la función sort() deben devolver 1 en caso de que el elemento b deba ir antes que el elemento a y vice versa, si se devuelve 0 se entiende que el orden entre ambos elementos es indiferente. De esta manera, la función devuelve -1 si la dirección de inicio de a es menor que la de b porque el objetivo de la función es ordenar las particiones en función creciente de sus direcciones de inicio

createLists()

```
// Creates a list for each instant
function createLists() {
  for (let instant of instants) {
    ul = document.createElement("ul");

    // To swipe between lists, the posterior to the first one come from the bottom, so they are .hidden. The .list c
    ul.classList.add("list", "hidden");
    // And the ones that go before the first one comes from the top, so they are already .gone, the animation of goi
    if (instant.instant < initialInstant)
      ul.classList.add("show", "gone");

    let index = 0;   // z-index of each new list item, decreases for every new one
    next_color = 0;  // Color for each new li, loops through an array of colors and always start for the same one

    for (let partition of instant.memory) {
      // When hovering, apart from the name, the address will be shown, in case that the partition only occupies one
      // it's a nonsense showing startAddress - endAddress so it only shows the startAddress
      let address = " " + partition.startAddress + (partition.getSize() == 1 ? "":" - " + partition.endAddress) ;
      if (partition instanceof EmptyPartition)
        addElement("EMPTY", "EMPTY" + address, index, ul, true);
      else
        addElement(partition.name, partition.name + address, index, ul);
      index--;
    }
    // Finally its title is set to its identifier as instant
    ul.children[0].style.setProperty("--title", "'instant " + instant.instant +"'");
    lists_container.appendChild(ul);
  }
  // Lists pushes every list created
  lists.push(...document.getElementsByClassName("list"));
}
```

No profundizaremos en esta función. Resumiendo, la función añade nuevas listas cuyos elementos tienen un índice que va decreciendo y un color que va cambiando a través de un array. En las listas, nombre del proceso o empty en caso de que esté vacía, se muestra por defecto. No obstante, cuando el ratón se pasa por encima del elemento el texto cambia a "Proceso1 0-99" por ejemplo, suponiendo que Proceso1 es el nombre de la partición y requiera 100 de memoria. En caso de que el tamaño de la partición sea de uno, en lugar de mostrar "Proceso1 0-0" se muestra "Proceso1 0".

Download()

Esta función se encuentra en el archivo thirdScreenActions.js

```
let text;

function download() {
  // Generates the text that the download file will have. The code could have been abbreviated
  if (!text) { // Once generated, for next downloads there's no need to generate it again
    text = "";
    for (let instant of instants) {
      text += instant.instant;
      text += ' ';
      for (partition of instant.memory) {
        text += '[';
        text += partition.startaddress;
        text += ' ';
        text += partition instanceof EmptyPartition ? 'hueco':partition.name;
        text += ' ';
        text += partition.getSize();
        text += '] ';
      }
      text += ' \n';
    }
  }

  // Creates an anchor element, which links to a download with the text previously generated.
  let element = document.createElement('a');
  element.setAttribute('href', 'data:text/plain;charset=utf-8,' + encodeURIComponent(text));
  element.setAttribute('download', 'gestormemoria.txt');

  element.style.display = 'none';
  document.body.appendChild(element);

  // Automatically clicks the button and erases it
  element.click();

  document.body.removeChild(element);
}
```

NOTA: el operador ? es una abreviación de un if, de manera que en los siguientes ejemplos son equivalentes: true?text="hola":text="adiós";

if (true)
text="hola";
else
text="adiós";

Cabe recalcar que la función se podría haber acortado. No obstante, hemos elegido dejarla así para que tenga una mayor legibilidad. La variable text es global para que una vez que acabe la función no se pierda su valor y no se tenga que generar por segunda vez si vuelven a llamar a la función, ya que una vez seleccionado el archivo definitivo no se puede cambiar y, por tanto, el

texto generado tampoco. Después de generar el texto se crea un link de descarga para un archivo “gestormemoria.txt” que lo contiene y automáticamente se activa, una vez iniciada la descarga se elimina el link de la página.