

# SISTEMAS OPERATIVOS

## Práctica 3

Gestión de memoria.

Alumno: Elvi Mihai Sabau Sabau.

DNI: 51254875L.

## Enunciado:

Se pretende realizar una simulación de gestión de memoria con particiones dinámicas. Se ofrecerá un menú con las dos opciones o algoritmos de asignación de particiones a implementar: peor hueco y mejor hueco.

El programa recibirá como entrada un archivo que tendrá una línea por cada proceso a cargar con el formato siguiente: <Proceso> <Instante\_llegada> <Memoria\_requerida> <Tiempo\_de\_ejecución>. La cantidad de memoria total será de 2000 y la unidad de asignación mínima será 100. El resultado, como mínimo, será guardado en un fichero "particiones.txt", donde se representarán, en líneas de texto, el estado de la memoria con cada entrada o salida de procesos, con el formato:

Instante de tiempo [Dirección\_inicio\_1 Estado\_1 tamaño\_1] ... [Dirección\_inicio\_n Estado\_n tamaño\_n] salto de línea

Por **ejemplo**:

1 [0 P1 300] [300 P2 200] [500 hueco 1500] salto de línea

La línea anterior indica que en el instante 1 tenemos dos particiones asignadas y un hueco. La primera partición empieza en la dirección 0, está ocupada por el proceso P1 y ocupa 300. La segunda partición empieza en la dirección 300, está ocupada por el proceso P2 y su tamaño es 200. Finalmente, tenemos un hueco que empieza en la dirección 500 y su tamaño es de 1500. Obviamente, la suma de 300, 200 y 1500 se corresponde con los 2000 que tenemos de memoria.

## Lenguaje usado:

Yo usaré [typescript](#), en el entorno de [Deno](#), ya que estoy más acostumbrado a este lenguaje de programación que otro cualquiera, además, creo que será más fácil manejar los datos con este lenguaje que con otros.

## Planteamiento:

Debido a que lo que tenemos son elementos que sacamos y metemos independientemente de la posición, lo primero que se me ocurrió fue hacer una lista doblemente enlazada, ya que de esta manera será más fácil manejar los procesos, la memoria será nuestra lista con un puntero de principio - fin, y los procesos serán los nodos, cada nodo tendrá un puntero a su siguiente nodo y a su anterior, además de los datos.

Originalmente, los procesos estarán alojados en un array de nodos ordenado descendentemente por tiempo de llegada, y después, en el bucle general, si el tiempo de llegada del último

elemento del array es igual a la iteración actual del bucle, hacemos un pop y añadimos dicho nodo a nuestra memoria. Para manejar los tiempos de salida, cada nodo tendrá una propiedad que será su tiempo de vida, por cada iteración, a todos los nodos dentro de la memoria se le restará 1 a su tiempo de vida, y si su tiempo de vida es 0, dicho nodo será eliminado.

Para gestionar la memoria restante, la propia lista doblemente enlazada tendrá su propiedad de memoria, y cada vez que se añada o libere un proceso, se restará / sumará el peso del proceso a la memoria.

Los espacios también serán nodos especiales que podrán agruparse si están uno al lado de otro al liberar un proceso, como cuando dos gotas de agua se juntan.

## Código:

```
class ProcessNode {
  public next: null | ProcessNode = null;
  public prev: null | ProcessNode = null;
  public start: number = 0;
  public end: number = 0;
  public empty: boolean = true;
  public aliveTime: number = 0;

  constructor(
    public process: string,
    public memory: number,
    public arrivalTime: number,
    aliveTime?: number,
    start?: number,
    end?: number,
  ) {
    if (start) this.start = start;
    if (end) this.end = end;
    if (process !== "HUECO") this.empty = false;
    if (aliveTime) this.aliveTime = aliveTime;
  }

  public turnIntoProcess(node: ProcessNode): boolean {
    if (this.empty && !node.empty) {
      this.process = node.process;
      this.arrivalTime = node.arrivalTime;
      this.empty = false;
      return true;
    }
  }

  console.log(
    "turnIntoProcess() => Receiving node empty or current node is not empty..",
  );
  return false;
}

public remove(): ProcessNode | boolean {
  if (!this.empty) {
    this.empty = true;
    this.process = "HUECO";

    return this.merge();
  }
}
```

```
}

console.log("remove() => Failed removing node, node already empty.");
return false;
}

public tick(): boolean {
    if (this.empty) return false;
    if (!this.empty) this.aliveTime--;
    if (this.aliveTime == 0) return true;
    return false;
}

private merge(): ProcessNode {
    // Si hay un hueco a la izquierda, se lo come y se vuelve uno.
    if (this.prev && this.prev.empty) {
        this.start = this.prev.start;
        this.memory += this.prev.memory;

        if (this.prev.prev) {
            this.prev.prev.next = this;
            this.prev = this.prev.prev;
        } else this.prev = null;
    }

    // Si hay un hueco a la derecha, se lo come y se vuelve uno.
    if (this.next && this.next.empty) {
        this.end = this.next.end;
        this.memory += this.next.memory;

        if (this.next.next) {
            this.next.next.prev = this;
            this.next = this.next.next;
        } else this.next = null;
    }

    return this;
}

public toString(): String {
    return `[ ${this.start} ${this.process} ${this.memory} ]`;
}
}

class Memory {
    public first: null | ProcessNode = null;
    public last: null | ProcessNode = null;

    constructor(public memory: number) {
        this.first = this.last = new ProcessNode(
            "HUECO",
            this.memory,
            0,
            0,
            0,
            memory - 1,
        );
    }

    public toString(): string {
        let msg = "";
        let current: null | ProcessNode = this.first;
```

```
while (current != null) {
    msg += current.toString();
    current = current.next;
}

return msg;
}

public worst(node: ProcessNode): boolean {
    let biggest: ProcessNode = new ProcessNode("COMPARISON_NODE", 0, 0);
    let current: null | ProcessNode = this.first;

    // Busca el hueco más grande.
    while (current != null) {
        if (
            (current.empty) &&
            (current.memory >= biggest.memory)
        ) {
            biggest = current;
        }

        current = current.next;
    }

    console.assert(
        this.insertIntoGap(biggest, node),
        "worst() => Failed inserting into gap",
    );
    return false;
}

public best(node: ProcessNode): boolean {
    let fittest: ProcessNode = new ProcessNode("COMPARISON_NODE", Infinity, 0);
    let current: null | ProcessNode = this.first;

    while (current != null) {
        // Busca el más pequeño, pero en el que pueda caber el proceso.
        if (
            (current.empty) &&
            (current.memory <= fittest.memory) && // Si es más pequeño que el actual.
            (current.memory >= node.memory) // Si es suficientemente grande para el proceso.
        ) {
            fittest = current;
        }
        current = current.next;
    }

    console.assert(
        this.insertIntoGap(fittest, node),
        "best() => Failed inserting into gap",
    );
    return true;
}

public insertIntoGap(
    emptyNode: ProcessNode,
    processNode: ProcessNode,
): boolean {
    // Primero comprobamos el que el nodo realmente sea un nodo de un hueco y no de un proceso.
    // Despues comprobamos que el hueco es mayor o igual a la memoria del proceso.
    if (!emptyNode.empty) return false;
```

```
if (emptyNode.memory < processNode.memory) return false;

// Si ocupa lo mismo que el hueco exactamente, convierte el hueco en el proceso.
if (emptyNode.memory == processNode.memory) {
    return emptyNode.turnIntoProcess(processNode);
}

// Enlazamos, primero comprobamos si detras del hueco habia algo.
if (emptyNode.prev) {
    processNode.prev = emptyNode.prev;
    emptyNode.prev.next = processNode;
}

// Enlazamos los nodos entre si.
processNode.next = emptyNode;
emptyNode.prev = processNode;

if (this.first === emptyNode) this.first = processNode;

// Recalculamos el inicio y el final de cada.
processNode.start = emptyNode.start;
processNode.end = emptyNode.start + processNode.memory - 1;
emptyNode.start = processNode.end + 1;
emptyNode.memory -= processNode.memory;

this.memory -= processNode.memory;

return true;
}

public tick(): ProcessNode[] {
    let deadProcs: ProcessNode[] = [];
    let current: null | ProcessNode = this.first;

    while (current != null) {
        if (current.tick()) {
            deadProcs.push(
                Object.assign(Object.create(Object.getPrototypeOf(current)), current),
            );

            this.memory += current.memory;
            const vac: ProcessNode | boolean = current.remove();
            if (vac !== false && (vac as ProcessNode).start == 0) {
                this.first = vac as ProcessNode;
            }
        }

        current = current.next;
    }

    return deadProcs;
}

}

// -- Main --

async function main(args: any) {
    if (args.length < 1) return false;
    const rammemory: number = 2000;
    const RAM: Memory = new Memory(rammemory);
    let processes: ProcessNode[] = [];
```

```
(await Deno.readFile(args[0])).split("\n").forEach((txt_line) => {
  const procOnArr = txt_line.split(" ");
  processes.push(
    new ProcessNode(
      procOnArr[0],
      +procOnArr[2],
      +procOnArr[1],
      +procOnArr[3],
    ),
  );
});

processes.sort((b, a) => a.arrivalTime - b.arrivalTime);
// processes.map((proc) => console.log(proc.arrivalTime, proc.aliveTime));

let log: string = "";
let iter: number = 1;

if (args[1].toLowerCase() == "mejor") {
  log += "\n|-----MEJOR HUECO-----|\n";
} else {
  log += "\n|-----PEOR HUECO-----|\n";
}

console.log("Tick:", iter - 1, RAM.toString());
log += `Tick: ${iter - 1} ${RAM.toString()}\n`;

do {
  RAM.tick();

  while (
    args[1].toLowerCase() == "mejor" &&
    processes.length > 0 &&
    processes[processes.length - 1].arrivalTime == iter
  ) {
    RAM.best(processes.pop() as ProcessNode);
  }

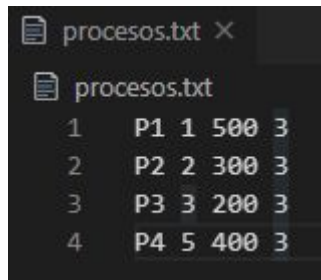
  while (
    args[1].toLowerCase() == "peor" &&
    processes.length > 0 &&
    processes[processes.length - 1].arrivalTime == iter
  ) {
    RAM.worst(processes.pop() as ProcessNode);
  }

  iter++;
  console.log("Tick:", iter - 1, RAM.toString());
  log += `Tick: ${iter - 1} ${RAM.toString()}\n`;
} while (RAM.memory !== rammemory);

Deno.writeFile("./particiones.txt", log);
}

main(Deno.args);
```

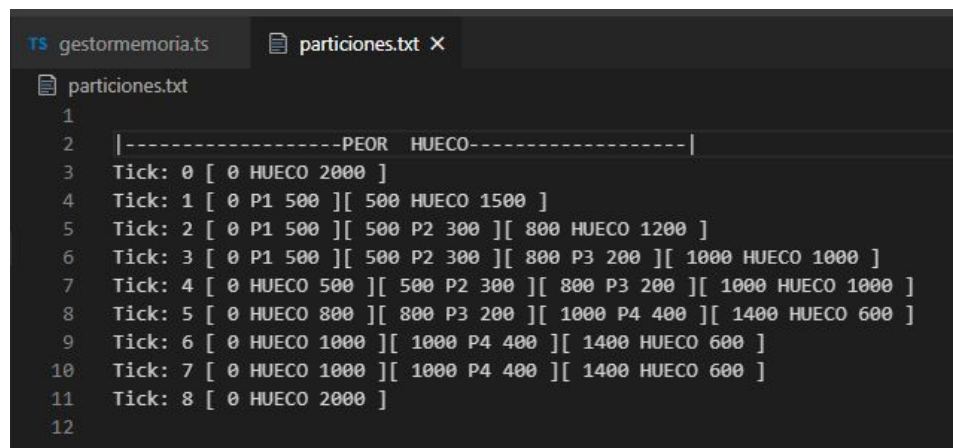
## Procesos a comprobar:



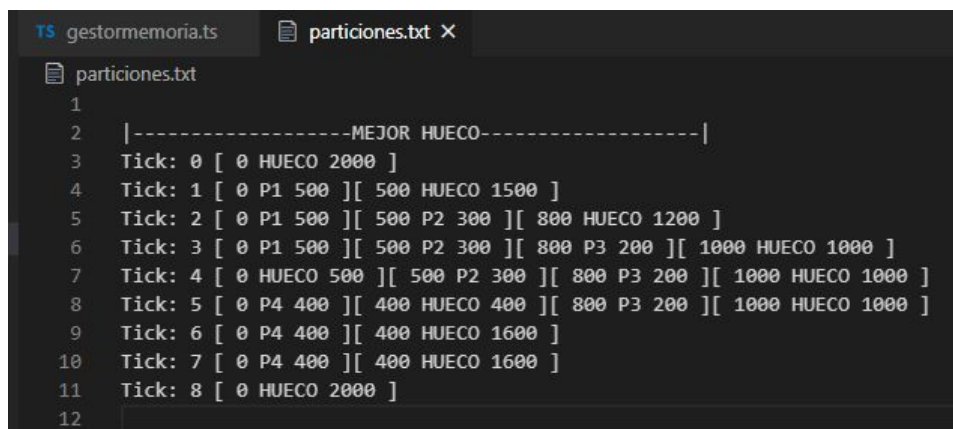
1	P1	1	500	3
2	P2	2	300	3
3	P3	3	200	3
4	P4	5	400	3

## Salidas:

```
PS C:\Users\Sapro\Desktop\UA\SISTEMAS OPERATIVOS\Prácticas SO 2020-2021> deno.exe run --allow-read --allow-write=particiones.txt .\gestormemoria.ts procesos.txt mejor
Tick: 0 [ 0 HUECO 2000 ]
Tick: 1 [ 0 P1 500 ][ 500 HUECO 1500 ]
Tick: 2 [ 0 P1 500 ][ 500 P2 300 ][ 800 HUECO 1200 ]
Tick: 3 [ 0 P1 500 ][ 500 P2 300 ][ 800 P3 200 ][ 1000 HUECO 1000 ]
Tick: 4 [ 0 HUECO 500 ][ 500 P2 300 ][ 800 P3 200 ][ 1000 HUECO 1000 ]
Tick: 5 [ 0 P4 400 ][ 400 HUECO 400 ][ 800 P3 200 ][ 1000 HUECO 1000 ]
Tick: 6 [ 0 P4 400 ][ 400 HUECO 1600 ]
Tick: 7 [ 0 P4 400 ][ 400 HUECO 1600 ]
Tick: 8 [ 0 HUECO 2000 ]
PS C:\Users\Sapro\Desktop\UA\SISTEMAS OPERATIVOS\Prácticas SO 2020-2021> deno.exe run --allow-read --allow-write=particiones.txt .\gestormemoria.ts procesos.txt peor
Tick: 0 [ 0 HUECO 2000 ]
Tick: 1 [ 0 P1 500 ][ 500 HUECO 1500 ]
Tick: 2 [ 0 P1 500 ][ 500 P2 300 ][ 800 HUECO 1200 ]
Tick: 3 [ 0 P1 500 ][ 500 P2 300 ][ 800 P3 200 ][ 1000 HUECO 1000 ]
Tick: 4 [ 0 HUECO 500 ][ 500 P2 300 ][ 800 P3 200 ][ 1000 HUECO 1000 ]
Tick: 5 [ 0 HUECO 800 ][ 800 P3 200 ][ 1000 P4 400 ][ 1400 HUECO 600 ]
Tick: 6 [ 0 HUECO 1000 ][ 1000 P4 400 ][ 1400 HUECO 600 ]
Tick: 7 [ 0 HUECO 1000 ][ 1000 P4 400 ][ 1400 HUECO 600 ]
Tick: 8 [ 0 HUECO 2000 ]
PS C:\Users\Sapro\Desktop\UA\SISTEMAS OPERATIVOS\Prácticas SO 2020-2021>
```



```
TS gestormemoria.ts  particiones.txt X
particiones.txt
1
2 |-----PEOR HUECO-----|
3 Tick: 0 [ 0 HUECO 2000 ]
4 Tick: 1 [ 0 P1 500 ][ 500 HUECO 1500 ]
5 Tick: 2 [ 0 P1 500 ][ 500 P2 300 ][ 800 HUECO 1200 ]
6 Tick: 3 [ 0 P1 500 ][ 500 P2 300 ][ 800 P3 200 ][ 1000 HUECO 1000 ]
7 Tick: 4 [ 0 HUECO 500 ][ 500 P2 300 ][ 800 P3 200 ][ 1000 HUECO 1000 ]
8 Tick: 5 [ 0 HUECO 800 ][ 800 P3 200 ][ 1000 P4 400 ][ 1400 HUECO 600 ]
9 Tick: 6 [ 0 HUECO 1000 ][ 1000 P4 400 ][ 1400 HUECO 600 ]
10 Tick: 7 [ 0 HUECO 1000 ][ 1000 P4 400 ][ 1400 HUECO 600 ]
11 Tick: 8 [ 0 HUECO 2000 ]
12
```



```
TS gestormemoria.ts  particiones.txt X
particiones.txt
1
2 |-----MEJOR HUECO-----|
3 Tick: 0 [ 0 HUECO 2000 ]
4 Tick: 1 [ 0 P1 500 ][ 500 HUECO 1500 ]
5 Tick: 2 [ 0 P1 500 ][ 500 P2 300 ][ 800 HUECO 1200 ]
6 Tick: 3 [ 0 P1 500 ][ 500 P2 300 ][ 800 P3 200 ][ 1000 HUECO 1000 ]
7 Tick: 4 [ 0 HUECO 500 ][ 500 P2 300 ][ 800 P3 200 ][ 1000 HUECO 1000 ]
8 Tick: 5 [ 0 P4 400 ][ 400 HUECO 400 ][ 800 P3 200 ][ 1000 HUECO 1000 ]
9 Tick: 6 [ 0 P4 400 ][ 400 HUECO 1600 ]
10 Tick: 7 [ 0 P4 400 ][ 400 HUECO 1600 ]
11 Tick: 8 [ 0 HUECO 2000 ]
12
```