Programación de Sistemas y Concurrencia (Systems Programming and Concurrency)

Lab nº1. Dynamic Memory Management

We have to develop a program to manage the memory of a drone (UAV – Unmanned Aerial Vehicle) that takes photos in order to detect cracks and deformations in bridges and buildings. This drone has two types of memory: i) main memory (RAM), and ii) fast cache memory where the photos are stored just when they are taken. Each image has a different size, and this problem focuses on the management of the cache memory to obtain the best performance.

This memory is an addressing space which starts in address `0` and ends in address `MAX=99` (this number is a constant). We will manage dynamically this amount of memory by allocating and deallocating contiguous parts on it. To do this we will use a list whose basic structure is:

```
typedef struct T_Node* T_handler;
struct T_Node {
      unsigned start;
      unsigned end;
      T_handler next;
};
```

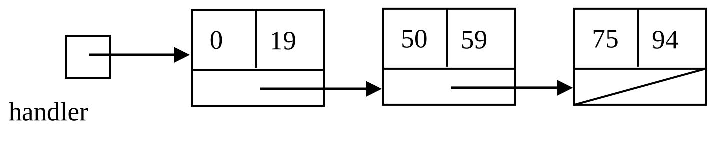
In this definition, each node refers to a contiguous range of free memory (from address `start` up to address `end`, both included) available in the whole amount of memory (see figure). At the very beginning there is a single node whose fields contain the values `start = 0` and `end = MAX`. In addition, the sequence of nodes of this list is ascendingly ordered by the value of the field `start`. Only the nodes that represent contiguous free memory are stored.

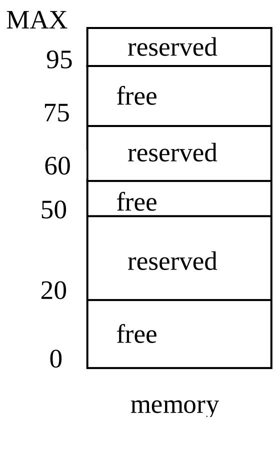
When the drone takes an image of length *size*, we have to allocate a contiguous block of memory to store it (**Allocate** operation). To do this, we have to look for a node large enough, and the fields of such node must be modified to reflect that it represents now a lower amount of free memory. If the required amount of memory (*size*) is exactly the same as the available in a node, then that node is removed from the list.

However, the images are not stored in cache forever but they are sent to a remote server via GSM (and, depending on the phone coverage, a photo may be sent complete or in several parts). Therefore, when the image (or a part of it) is sent, we want to deallocate the piece of memory of length *size* where it is stored (**Deallocate** operation). To do this, we add to the list a new node with the required information into its fields `start` and `end`. Afterwards, it may happen that the just inserted node represents a piece of memory next to an already existing one so, in such a case, we have to put them together into a single one. We must have this in mind in order to avoid nodes referencing contiguous pieces of memory, i.e. two consecutive nodes where the `end` field of the first one matches with the field `start` (minus one) of the second node. This procedure is known as compacting memory (two, or even three, nodes are compacted into only one).

Develop the next functions declared in the header file `MemoryManager.h`:

```
// Creates the required structure to manage the available memory
void Create(T_handler* handler);

// Free the required structure
void Destroy(T_handler* handler);

/* Returns in "ad" the memory address where the required block
 * of memory with length "size" starts. If this operation finishes
 * successfuly then "ok" holds a TRUE value; FALSE otherwise.
 */
void  Allocate(T_handler*  handler,  unsigned  size,  unsigned*  ad,
unsigned* ok);

/* Shows the current status of the memory */
   void Show (T_handler handler);

/* Frees a block of memory with length "size" which starts at "ad"
 * address.  If needed, can be assume to be a previous allocated
 * block
 */
void Deallocate(T_handler* handler, unsigned size, unsigned ad);
```


0
19
50
59
75
94
handler


MAX
95
75
60
50
20
0
reserved
free
reserved
free
reserved
free
memory