Realizzazione di un gestore di heap per allocazione di memoria dinamica

L'esercizio proposto affronta un esempio di allocazione di memoria contigua nell'ambito di un processo in esecuzione. **NON si tratta quindi di una vera e propria componente del sistema operativo**, quale ad esempio il gestore della memoria da allocare ai vari processi. Lo si propone in quanto le problematiche sono simili e il contesto risulta semplificato.

Si ricorda che un heap costituisce un'area di memoria resa disponibile a un programma mediante primitive di allocazione quali malloc/calloc e free. Queste ultime agiscono su di un heap default (a cui non si fa quindi riferimento esplicito), mentre l'esercizio proposto fa riferimento a un heap esplicito. PER SEMPLICITA', la memoria virtuale gestita dall'heap sarà allocata una tantum utilizzando la funzione malloc.

Un programma, dopo aver creato un heap `h` mediante `heapCreate`, potrà quindi allocare memoria su questo heap, mediante istruzioni del tipo

```
V = heapAlloc(h,n*sizeof(int));
```

anziché

```
V = malloc(n*sizeof(int));
```

In pratica, si alloca una volta sola (mediante una `malloc` all'interno della `heapCreate`), una zona di memoria contigua (detta heap), nella quale si vanno di volta in volta a servire richieste di allocazione, mediante ricerca di intervalli di memoria contigua.

Per ottenere questo risultato, occorrerà allocare, oltre all'area di memoria contigua (area per dati), altre strutture dati (di controllo), quali ad esempio: una struct puntata, che realizzi l'oggetto/ADT heap, e (all'interno dell'heap), delle liste o tabelle in grado di gestire gli intervalli (o partizioni) di memoria allocati e quelli liberi.

Si vuole quindi realizzare un modulo di libreria utente in linguaggio C, in grado di fornire funzioni di gestione di un HEAP, esportato secondo il file header che segue:

```
/*-----------------------------------------------------------------
--------*/

/* myheap.h */

typedef enum {
  first_fit,
  best_fit,
  worst_fit
} heap_policy_t;

typedef struct heap_s heap_t;
```

```
heap_t *heapCreate (size_t heapSize);
void heapDestroy (heap_t *h);
void *heapAlloc (heap_t *h, size_t size);
void heapFree (heap_t *h, void *p);
void heapSetPolicy (heap_t *h, heap_policy_t policy);
void heapDefrag (heap_t *h);
void *heapNewPointer (heap_t *h, void *p);
void heapDefragClose (heap_t *h);

/*-----------------------------------------------------------------
--------*/
```

Le funzioni proposte costituiscono quindi un'alternativa all'utilizzo diretto di `malloc/free` (alle quali tuttavia si fa riferimento internamente), quali sono fornite dalla libreria C.
Si noti che **NON si chiede di realizzare un meccanismo di gestione di indirizzi logici/fisici**, ma si lavora unicamente nello spazio degli indirizzi virtuali/logici di un processo.
L'esercizio, in sostanza, fornisce l'opportunità di sperimentare le tecniche di allocazione contigua e di deframmentazione.

Il modulo deve permettere di creare o distruggere (mediante le funzioni `heapCreate` e `heapDestroy`) un heap, cioè un'area contigua in memoria, nella quale sia possibile effettuare operazioni di allocazione analoghe a `malloc` e `free`, mediante le funzioni `heapAlloc` e `heapFree`.

La struct `heap_t` contiene tutte le informazioni necessarie a gestire un heap:

- L'area dati contigua (ad esempio un puntatore a void (meglio) o a char); questo puntatore può essere utile in quanto rappresenta l'intera area, allocata dalla `heapCreate`, in cui si vanno a cercare gli intervalli da allocare. Può tuttavia essere considerato opzionale, poiché replica un'informazione già rappresentata dalla prima partizione libera.
- La lista delle partizioni libere (contigue) nell'heap (inizialmente tutta la memoria dell'heap, allocata dall'heapCreate, mediante malloc).
- La lista (o tabella) delle partizioni di memoria allocate.
- Altre eventuali informazioni necessarie per la gestione (es. politica di ricerca di partizione libera).

Breve descrizione delle funzioni:

- `heapCreate` e `heapDestroy`: ottengono un'area contigua di memoria, e la rilasciano al termine, utilizzando malloc (o funzione equivalente, es. calloc) e free. Una volta ottenuto un heap, tutte le allocazioni e i rilasci di memoria che fanno riferimento a tale heap sono effettuati nell'area di memoria ottenute con `heapCreate`. Tutte le dimensioni vanno intese in byte.
- `heapAlloc`: cerca (e restituisce, se trovato) un intervallo contiguo di memoria, tra quelli disponibili (è necessaria una free list delle partizioni libere).
- `heapFree`: libera un intervallo contiguo di memoria, a partire dal puntatore. Il puntatore va cercato in una lista o tabella delle partizioni allocate, da cui la partizione va eliminata e aggiunta alla free list (tenendo conto di eventuali ri-compattazioni di partizioni adiacenti).
- `heapSetPolicy` imposta la politica di ricerca di una partizione libera.

(parte opzionale, importante ma da realizzare dopo aver consolidato e testato le funzionalità precedenti)

- `heapDefrag` ricompatta (de-frammenta) le partizioni occupate (copiandone i contenuti dalla posizione originale a quella nuova). DOPO LA RICOMPATTAZIONE, I PUNTATORI PRECEDENTEMENTE RILASCIATI NON SARANNO PIU' VALIDI (in quanto non si gestiscono puntatori logici/fisici). Quindi si fornisce un'apposita funzione (`heapNewPointer`), che permetta di ottenere il nuovo puntatore a una partizione, dopo de-frammentazione, a partire da quello vecchio: occorre predisporre internamente una opportuna tabella o lista di conversione. Tale tabella deve essere generata dopo ogni `heapDefrag`, e cancellata (o svuotata) ad una successiva chiamata esplicita di `heapDefragClose`.
  Un programma che chiami `heapDefrag` dovrebbe quindi, dopo la de-frammentazione, aggiornare tutti i suoi puntatori, utilizzando `heapNewPointer`, e infine chiamare `heapDefragClose`.

Al fine di fare un test dell'allocatore proposto, si consiglia di utilizzare un client *(se ne fornisce un semplice esempio)* che, dato un file testo in ingresso (visto come sequenza di righe terminate da '\n'), e tre numeri D, N e M (ricevuti in ingresso), svolga le seguenti operazioni:

- Generazione di un heap di dimensione D; apertura del file testo in ingresso; apertura di un secondo file in uscita.
- Lettura di pagine di N righe successive dal file testo, da acquisire in un vettore S di N stringhe (di lunghezza variabile). Sia il vettore di puntatori che le stringhe vanno allocati dinamicamente nell'heap precedentemente creato.
    - Ordinamento delle stringhe nel vettore, mediante un algoritmo di sorting, che agisca sui puntatori.
    - Le stringhe di dimensione maggiore di M (acquisito in input) vanno copiate in un secondo vettore di stringhe T (sovra-dimensionato ed eventualmente ri-allocato).
    - Il vettore S (ordinato) va copiato nel file aperto in output.
- Al termine anche il vettore T va ordinato e scritto in un terzo file in output.

parte opzionale, non presente nel client proposto) Qualora non sia possibile allocare una stringa (a causa di frammentazione), è necessario fare de-frammentazione e aggiornare tutti i puntatori.

SI CONSIGLIA DI PROVARE IL CLIENT CON MALLOC/FREE, prima di passare al proprio allocatore.