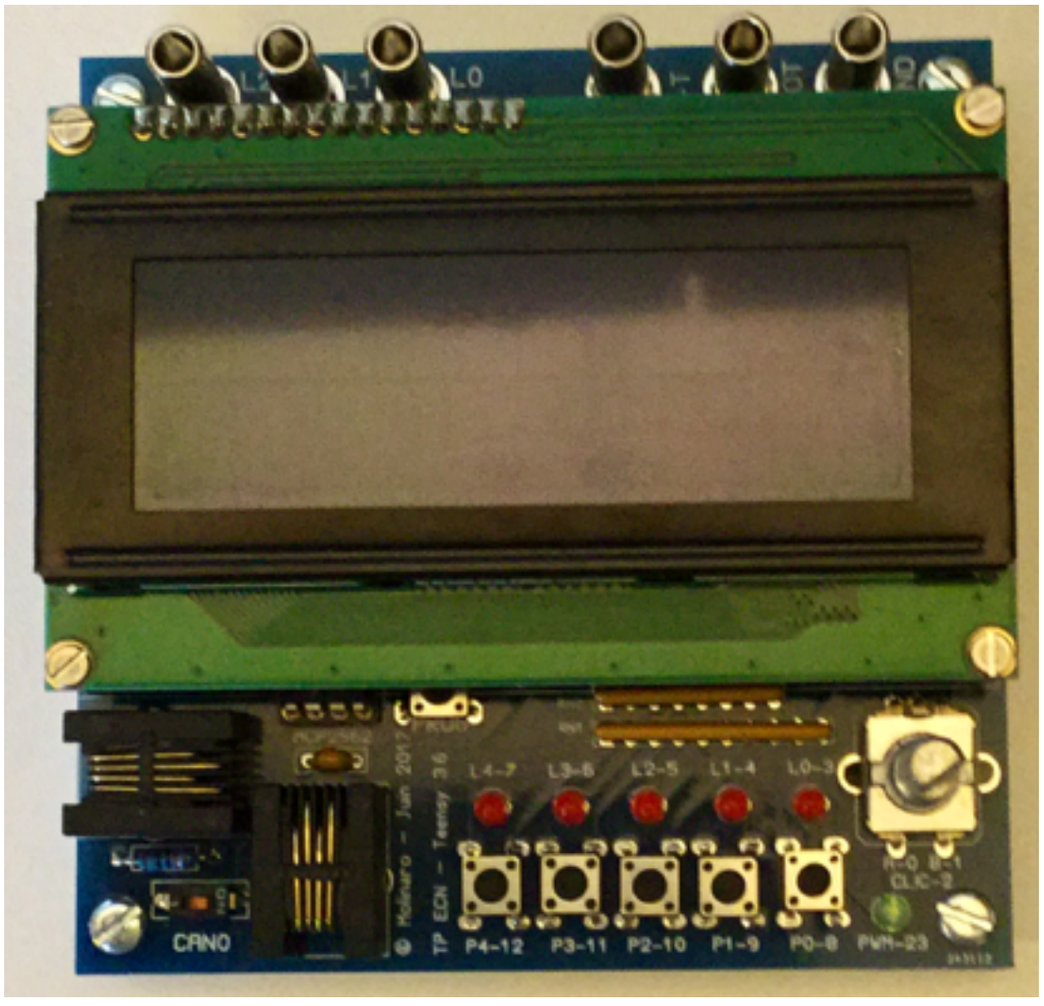
Temps Réel



Programme *17-dynamic-allocation*

Pierre Molinaro
Juin 2018

Description de cette étape

Jusqu'à présent, l'allocation dynamique n'était pas nécessaire pour l'exécutif.

L'implémentation des commandes gardées de l'étape suivante utilise l'allocation dynamique (il est possible d'adopter une implémentation des commandes gardés sans allocation dynamique, il faut simplement allouer statiquement des tableaux de taille suffisante).

Dans la suite, on présente les principales fonctions d'allocation dynamique en C et C++, puis comment elles sont réalisées dans l'exécutif.

CENTRALE NANTES

Allocation dynamique en C

Vu du programmeur C, l'allocation dynamique se résume à ces deux principales fonctions :

- **malloc**, pour allouer la mémoire;
- **free**, pour la libérer.

Ces fonctions sont implémentées par la librairie C. Sous Unix, les commandes **man alloc** et **man free** permettent d'afficher leur description :

```
void * malloc(size_t size);
```

The malloc() function allocates size bytes of memory and returns a pointer to the allocated memory.

```
void free(void *ptr);
```

The free() function deallocates the memory allocation pointed to by ptr. If ptr is a NULL pointer, no operation is performed.

Allocation dynamique en C++

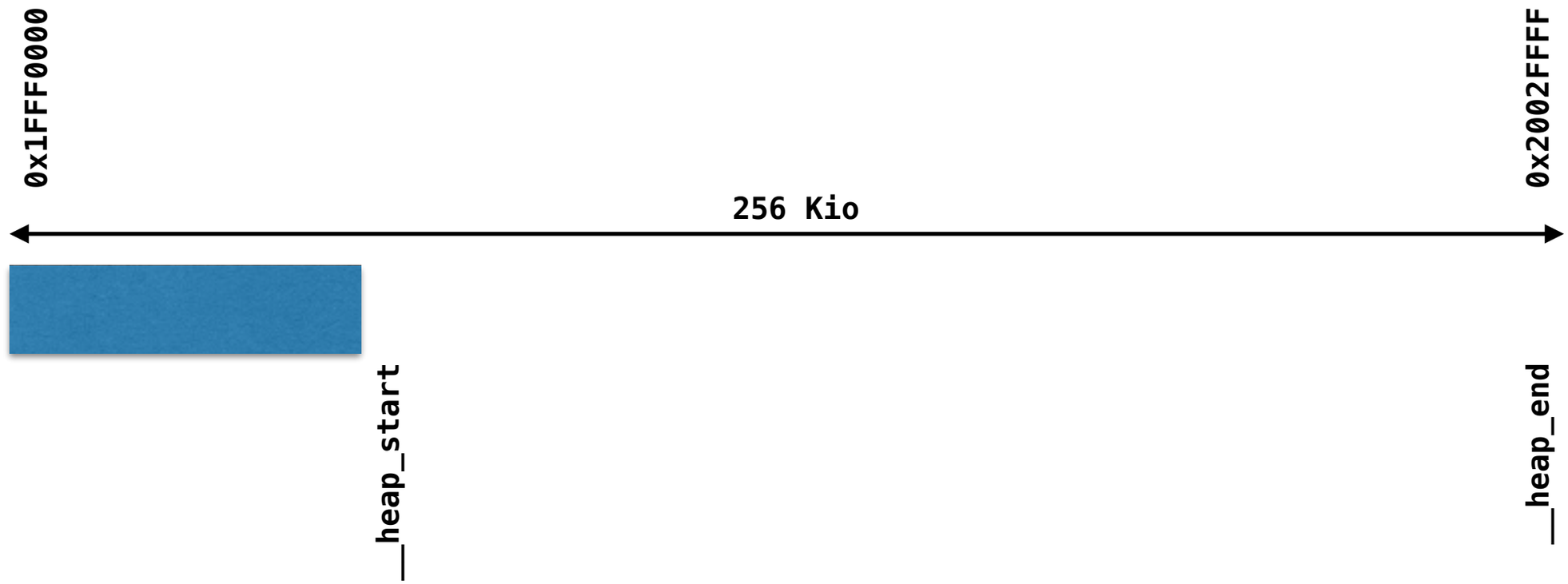
L'allocation dynamique en C++ s'exprime par les constructions **new** et **delete** :

- **new *type***, pour allouer un objet ;
- **new *type [taille]***, pour allouer un tableau d'objets ;
- **delete ptr**, pour libérer la mémoire associée à un objet ;
- **delete [] ptr**, pour libérer la mémoire associée à un tableau d'objets.

Ces constructions sont définies dans le langage C++ (à la différence du C), et on verra comment ce langage permet de rediriger les allocations et les libérations vers les fonctions C correspondantes.

CENTRALE NANTES

Utilisation de la mémoire du micro-contrôleur



Le micro-contrôleur MK66FX1M0 qui équipe le module Teensy 3.6 intègre une RAM de 256 Kio, aux adresses `0x1FFF0000` à `0x2002FFFF`.

Le début de la mémoire est utilisé par les variables globales et les piles des tâches.

Le reste est le **tas** (« heap »), c'est-à-dire l'espace disponible pour l'allocation dynamique. L'éditeur des liens maintient deux symboles qui délimitent le tas, **__heap_start** et **__heap_end**. Vous pouvez connaître la valeur de ces symboles en consultant pour chaque programme le fichier **zPRODUCTS/product.map**.

CENTRALE NANTES

Gestion du tas

Il existe de nombreuses politiques de gestion de la mémoire, nous allons adopter la plus simple, qui consiste à gérer des blocs de taille fixée.

Un certain nombre de tailles est choisi, en progression suivant les puissances de 2 : par exemple, 8, 16, 32, 64, 128, 256. Le plus grand bloc allouable aura une taille de 256 octets.

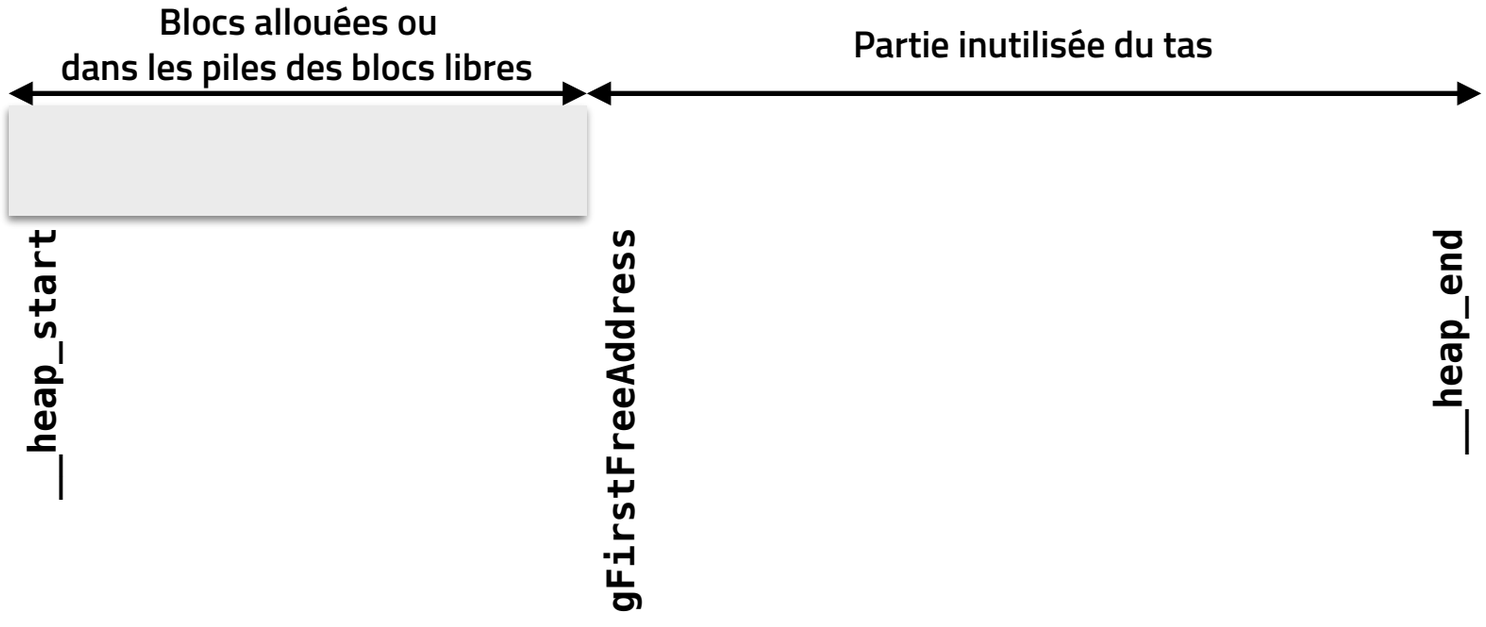
À chaque taille, correspond une pile de blocs libres. Initialement, toutes ces piles sont vides.

Allocation : la taille demandée est arrondie à la puissance de 2 supérieure. Si il y a un bloc libre dans la pile correspondante, il en est retiré. Sinon, le bloc est alloué dans la partie inutilisé du tas.

Libération : le bloc est ajouté à la pile libre correspondant à sa taille.

Lien :

https://en.wikipedia.org/wiki/Memory_management



L’archive

L’archive **17-files_tar.bz2** contient les fichiers **heap.h** et **heap.cpp** qui implémentent les routines de gestion de la mémoire.

Dans la suite, nous allons décrire les définitions et fonctions disponibles.

Les piles de blocs libres

À la fin du fichier **heap.h** , deux définitions fixent les tailles allouables :

```
// Biggest block being allocated = 2 ** kMaxSizePowerOfTwo
static const size_t kMaxSizePowerOfTwo = 10
// Smallest block being allocated = 2 ** kMinSizePowerOfTwo (SHOULD BE >= 3)
static const size_t kMinSizePowerOfTwo =  3 ;
```

C’est-à-dire que les valeurs ci-dessus permettent d’allouer des blocs de 8, 16, 32, 64, 128, 256, 512 ou 1024 octets. La demande d’allocation d’une taille supérieure renverra la valeur **nullptr**.

CENTRALE NANTES

Allocation et concurrence

Les fonctions d'allocation et de libération peuvent être appelées en concurrence par différentes tâches et par des routines d'interruption. Il faut donc s'assurer que leur exécution soit exempt de concurrence.

Une solution simple est de les exécuter interruptions masquées, dans une *section*. Ainsi, le fichier **heap.h** déclare les prototypes suivants :

```
//$section memory.alloc
void * memoryAlloc (const size_t inBlockSize) asm ("memory.alloc") ;
void * section_memoryAlloc (SECTION_MODE_ const size_t inBlockSize) asm ("section.memory.alloc") ;

//$section memory.free
void memoryFree (void * inPointer) asm ("memory.free") ;
void section_memoryFree (SECTION_MODE_ void * inPointer) asm ("section.memory.free") ;
```

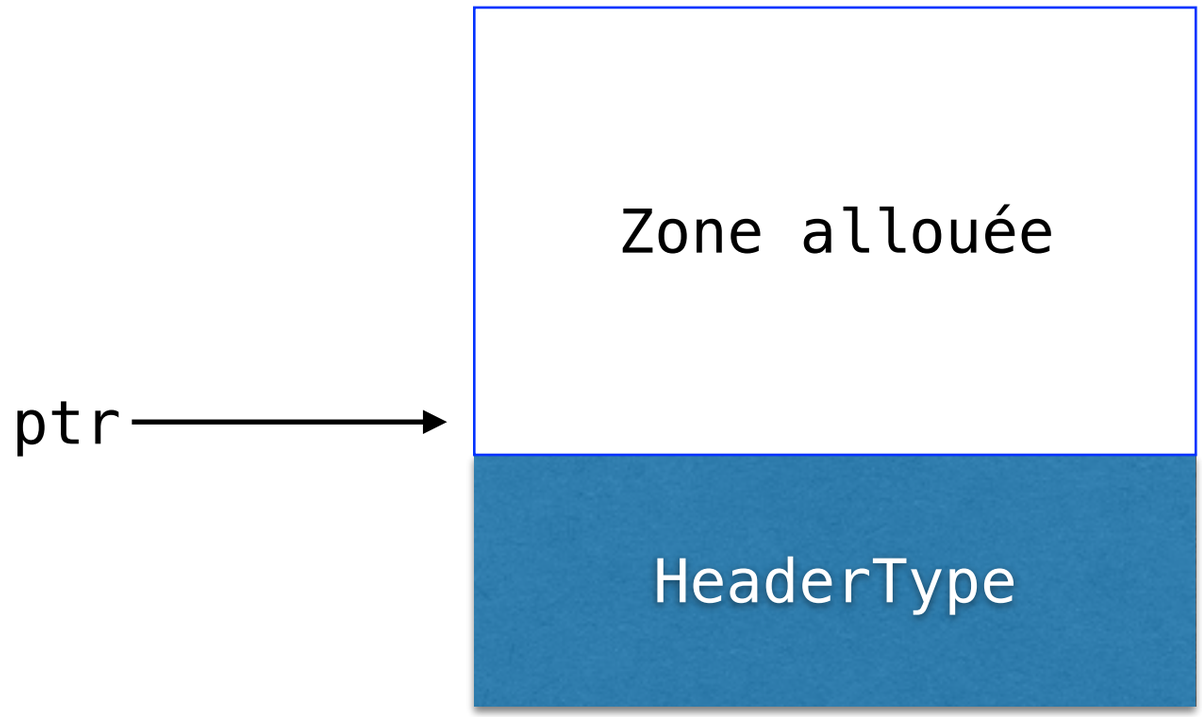
Remarquer que les prototypes des fonctions **memoryAlloc** et **memoryFree** n'ont pas d'annotation de mode, ce qui signifie qu'elles peuvent être appelées dans n'importe quel mode, tout en assurant que l'exécution des fonctions **section_...** s'effectuent interruptions masquées. On verra dans la suite que cela est indispensable pour prendre en charge l'allocation et la libération en C++.

Structure d’un bloc

Chaque bloc contient une en-tête de 8 octets, ce qui permet d’assurer que les adresses allouées sont des multiples de 8, ce qui est requis pour les entiers 64 bits.

```
typedef struct HeaderType {
    uint32_t mFreeListIndex ;
    HeaderType * mNextFreeBlock ;
} HeaderType ;
```

Note : on pourrait fusionner ces deux champs, il ne sont jamais utilisés simultanément.



Fonction section_memoryAlloc

```
void * section_memoryAlloc (SECTION_MODE_ const size_t inBlockSize) {
  HeaderType * result = nullptr ;
  if (inBlockSize > 0) {
  //--- Compute smallest block with size equal to a power of two bigger of equal to required size
    size_t smallestPowerOfTwo = 32 - (size_t) __builtin_clz (inBlockSize) ;
//     size_t v = inBlockSize - 1 ;
//     size_t smallestPowerOfTwo = 0 ;
//     while (v > 0) {
//       smallestPowerOfTwo ++ ;
//       v >>= 1 ;
//     }
  //--- Allocate if not too large
    if (smallestPowerOfTwo <= kMaxSizePowerOfTwo) {
      if (smallestPowerOfTwo < kMinSizePowerOfTwo) {
        smallestPowerOfTwo = kMinSizePowerOfTwo ;
      }
      const uint32_t freeListIndex = smallestPowerOfTwo - kMinSizePowerOfTwo ;
      tFreeBlockListDescriptor & descriptorPtr = gFreeBlockDescriptorArray [freeListIndex] ;
      if (descriptorPtr.mFreeBlockCount > 0) { // Allocate from free list
        descriptorPtr.mFreeBlockCount -- ;
        result = descriptorPtr.mFreeBlockList ;
        descriptorPtr.mFreeBlockList = result->mNextFreeBlock ;
        result ++ ;
        gCurrentlyAllocatedCount ++ ;
        gAllocationCount ++ ;
      }else{ // Allocate from heap
        result = (HeaderType *) gFirstFreeAddress ;
        const size_t size = (1U << smallestPowerOfTwo) + sizeof (HeaderType) ;
        gFirstFreeAddress += size ;
        if (gFirstFreeAddress >= (size_t) & __heap_end) { // Not enough space
          gFirstFreeAddress -= size ;
          result = nullptr ;
        }else{
          result->mFreeListIndex = freeListIndex ;
          result ++ ;
          gCurrentlyAllocatedCount ++ ;
          gAllocationCount ++ ;
        }
      }
    }
  }
  return result ;
}
```

Si un bloc de taille nulle est demandé, la valeur **nullptr** est retournée.

La fonction **__builtin_clz** retourne le nombre de bits consécutifs à zéros à partir du bit de poids fort.

Si un bloc de taille trop grande (> à 2** **kMaxSizePowerOfTwo**) est demandé, la valeur **nullptr** est retournée.

Si un bloc de taille < à 2** **kMinSizePowerOfTwo**) est demandé, le bloc alloué aura la taille 2** **kMinSizePowerOfTwo**.

La pile des blocs libres correspondant à la taille demandée n'est pas vide, on prélève un bloc dans cette pile.

La pile des blocs libres correspondant à la taille demandée est vide, on prélève un bloc dans la mémoire inutilisée. **gFirstFreeAddress** contient son adresse.

La mémoire inutilisée a une taille < à la taille demandée : retourner **nullptr**.

Fonction `section_memoryFree`

```
void section_memoryFree (SECTION_MODE_ void * inPointer) {
  if (nullptr != inPointer) {
    HeaderType * p = (HeaderType *) inPointer ;
    p -- ;
    const uint32_t idx = p->mFreeListIndex ;
    p->mNextFreeBlock = gFreeBlockDescriptorArray [idx].mFreeBlockList ;
    gFreeBlockDescriptorArray [idx].mFreeBlockList = p ;
    gFreeBlockDescriptorArray [idx].mFreeBlockCount ++ ;
    gCurrentlyAllocatedCount -- ;
  }
}
```

L'appel avec un argument null est sans effet.

Le bloc libéré est simplement empilé dans la pile des blocs libres correspondant à son indice mémorisé dans le champ **mFreeListIndex**.

CENTRALE NANTES

Allocation / libération en C++

Les constructions **new** et **delete** du C++ appellent des fonctions qui doivent être définies, et qui le sont le plus souvent dans la librairie standard. Ici, elles sont implémentées dans **heap.cpp**.

```
void * operator new (size_t inSize) {
  return memoryAlloc (inSize) ;
}

void * operator new [] (size_t inSize) {
  return memoryAlloc (inSize) ;
}

void operator delete (void * ptr) {
  memoryFree (ptr) ;
}

void operator delete [] (void * ptr) {
  memoryFree (ptr) ;
}
```

Voilà pourquoi il faut pouvoir appeler **memoryAlloc** et **memoryFree** sans annotation de mode : en effet, il n'y a aucun moyen de les faire apparaître dans les prototypes des fonctions requises par le C++.

Fonctions complémentaires...

Mais ce n'est pas tout : le runtime C++ a besoin que la variable **__dso_handle** et que les fonctions **__cxa_exit** et **__cxa_pure_virtual** soient définies.

```
void * __dso_handle ;

void __cxa_exit (void) ;

void __cxa_exit (void) {
  assertionFailure (0, __FILE__, __LINE__) ;
}

void __cxa_pure_virtual (void) ;

void __cxa_pure_virtual (void) {
  assertionFailure (0, __FILE__, __LINE__) ;
}
```

CENTRALE NANTES

Cas particulier : C++17

Et en C++17, deux autres fonctions doivent être implémentées. Elles sont simplement ignorées dans les versions précédentes de la norme du langage.

```
void operator delete (void * ptr, unsigned int) {
  memoryFree (ptr) ;
}

void operator delete [] (void * ptr, unsigned int) {
  memoryFree (ptr) ;
}
```

CENTRALE NANTES

Travail à faire

Imaginer un programme permettant de tester la gestion mémoire de façon intensive :

- la tâche de plus faible priorité boucle sur des allocations / libérations ;
- une autre tâche périodique effectue aussi des allocations / libérations ;
- afficher les statistiques d'allocations / libérations (voir le fichier **heap.h**) ;
- quand on appuie sur un bouton poussoir, les allocations cessent, les libérations s'effectuent : on doit aboutir un nombre nul d'objets alloués.

CENTRALE NANTES