

## Résumé

Les allocations mémoire sont généralement plus coûteuses que d'ordinaire en environnement *multi-thread*. En effet, la majorité des algorithmes d'allocation en usage dans les systèmes d'exploitation actuels implémentent le tas à l'aide d'un ou plusieurs *pool* de mémoire protégés par des mécanismes de synchronisation à exclusion mutuelle. Cela induit de la contention lorsque plusieurs *threads* ont besoin de mémoire simultanément. De plus, les primitives de verrouillage introduisent bien souvent une perte de performance même lorsqu'un seul processus les utilise.

À cela s'ajoute l'éventualité plus fourbe d'obtenir dans deux *threads* différents des blocs mémoire partageant la même ligne de cache processeur. Dans un tel cas de figure, si deux processeurs utilisent ces blocs mémoire indépendants simultanément, le mécanisme de cohérence du cache provoquera une importante perte de performance en le rafraîchissant à chaque accès.

Nous avons donc étudié la possibilité de limiter le nombre d'allocations concurrentes pour augmenter les performances tout en continuant d'utiliser l'allocateur du système à l'aide du mécanisme de «*Thread Local Storage*» (TLS).

## Première partie

# Le «*thread local storage*» et son utilisation

Le «*thread local storage*» est un procédé par lequel une variable globale ou statique, normalement unique dans l'espace d'adressage du processus, apparaît résider à une adresse différente pour chaque *thread*.

La variable globale `errno` est un exemple d'application typique du TLS : elle indique la dernière erreur ayant eu lieu et doit donc être différente pour chaque *thread*.

L'accès au «*thread local storage*» est généralement réalisé via des interfaces différentes suivant le système d'exploitation. Nous utiliserons ici l'interface POSIX, qui présente l'avantage d'être implémentée dans tous les systèmes d'exploitations majeurs.

## 1 Utilisation du TLS

En utilisant l'interface POSIX, il est nécessaire de déclarer la variable globale en tant que `pthread_key_t`, et de l'initialiser avec la fonction `pthread_key_create`. Chaque *thread* pourra ensuite associer un pointeur différent à cette variable globale et le retrouver à volonté. Un destructeur, associé à la variable lors de son initialisation, sera invoqué à sa destruction pour chaque pointeur enregistré.

### 1.1 Stocker et retrouver les données

Afin de pouvoir stocker des données variées, il est intéressant d'utiliser une structure pour les données du TLS.

Cette structure pourra être ainsi dynamiquement allouée et initialisée dans chaque *thread*, puis associée à la clé à l'aide de la fonction `pthread_setspecific`.

On pourra ensuite retrouver cette structure à tout moment en appelant la fonction `pthread_getspecific`.

### 1.2 Libération du TLS

La présence d'un destructeur indiqué à la création de la clé ne dispense pas de la gestion mémoire. Terminer les *threads* avec `pthread_exit` n'est pas suffisant, il convient de détruire la



clé avec `pthread_key_delete`. Cette fonction invoquera le destructeur pour libérer toutes les ressources associées à la clé.

## 2 Intérêt du TLS pour un cache d'allocation

On a vu précédemment que l'interface POSIX rend particulièrement aisée l'utilisation du TLS. Il devient ainsi très intéressant de l'utiliser pour associer à chaque *thread* d'exécution une réserve de blocs mémoire, dans laquelle on pourra puiser si nécessaire au lieu d'appeler l'allocateur système.

Le TLS étant spécifique à chaque *thread*, l'accès à cette réserve est très rapide car il ne nécessite aucun verrou. Le gain de performance est réalisé sur deux plans : en premier lieu, on évite d'appeler l'allocateur système pour récupérer directement un bloc mémoire ; en second lieu, on peut se passer de tout mécanisme de synchronisation, améliorant ainsi la concurrence de l'application.

Nous allons voir dans la partie suivante une application concrète de cette méthode.

## Deuxième partie

# Un exemple d'implémentation

Dans cet exemple concret, tiré de notre projet, nous utiliserons un type de données se prêtant particulièrement à cette optimisation : une encapsulation de chaîne de caractères.

Ce type de données est souvent de courte durée de vie, et par conséquent sujet à des allocations et libérations très fréquentes. De plus, dans le cadre de tâches répétitives, la taille des blocs alloués ne varie généralement que très peu.

Nous allons montrer qu'il est possible de diminuer le nombre d'allocations de plus de 50% dans une application complexe en utilisant un cache d'allocation en TLS.

## 1 Le cache

Le type sur lequel nous allons travailler encapsule une chaîne de caractères. Il gère également le découpage d'une chaîne en plusieurs sous-parties :

```
1 typedef struct m_string {
2     /* private */
3     struct m_string *parent; /* pointe sur la chaîne mere */
4     size_t _len;             /* longueur de la chaîne */
5     size_t _alloc;          /* taille du bloc memoire */
6     char *_data;            /* la chaîne elle meme */
7     uint16_t _flags;        /* options */
8
9     /* public (no more than 64k tokens) */
10    uint16_t parts;          /* nombre de sous-parties */
11    struct m_string *token; /* tableau de sous-parties */
12 } m_string;
```

Cette structure donne donc lieu à deux allocations différentes : une première de taille constante pour «l'enveloppe», la structure elle-même ; et une seconde de taille variable pour la chaîne encapsulée.

L'implémentation du cache sera très simple. Nous enregistrerons simplement les structures `m_string` dans un tableau, en laissant le bloc dédié au stockage de la chaîne intact. Les autres champs seront fixés à des valeurs par défaut inoffensives.

```
1 typedef struct _m_string_TLS {
2     unsigned int free_string;
3     m_string *string [STRING_CACHE_SIZE];
4 } _m_string_TLS;
5
6 static pthread_key_t TLS;
```

## 2 Initialisation et destruction du cache

Il convient en premier lieu d'initialiser la clé TLS :

```
1 int _string_TLS_setup(void)
2 {
3     if (pthread_key_create(& TLS, _string_TLS_destructor) == -1)
4         return -1;
5     return 0;
6 }
```

On remarque la référence au destructeur, défini comme suit :

```
1 static void _string_TLS_destructor(void *tls)
2 {
3     _m_string_TLS *tls_data = tls;
4
5     if (tls_data) {
6         while (tls_data->free_string --) {
7             free(tls_data->string[tls_data->free_string]->_data);
8             free(tls_data->string[tls_data->free_string]);
9         }
10        free(tls_data);
11    }
12 }
```

Nous devons ensuite initialiser les données du TLS, c'est à dire notre cache, pour les associer à la clé.

```
1 int _string_TLS_init(void)
2 {
3     _m_string_TLS *tls_data = malloc(sizeof(*tls_data));
4
5     if (! tls_data) return -1;
6
7     tls_data->free_string = 0;
8
9     if (pthread_setspecific(TLS, tls_data) != 0) {
10        free(tls_data);
11        return -1;
12    }
13
14    return 0;
15 }
```

Cette fonction, contrairement aux précédentes, devra être appelée par chaque *thread* souhaitant profiter du cache.

Quand tous les *threads* seront terminés, il nous faudra finalement supprimer la clé afin de libérer la mémoire :

```
1 void _string_TLS_cleanup(void)
2 {
3     pthread_key_delete(TLS);
4 }
```



### 3 Utilisation du cache pour optimiser les allocations

Voici le code de l'allocateur de la structure `m_string`, tirant parti du cache chaque fois que cela est possible.

```

1 m_string *string_alloc(const char *string, size_t len)
2 {
3     /** @brief allocate a m_string and initialize it with the given data */
4
5     m_string *new = NULL;
6     char *s = NULL;
7     _m_string_TLS *tls_data = pthread_getspecific(TLS);
8
9     if (tls_data->free_string) {
10        new = tls_data->string[--tls_data->free_string];
11    } else {
12        if (!(new = malloc(sizeof(*new))) ) {
13            perror(ERR(string_alloc, malloc));
14            return NULL;
15        }
16        new->_data = NULL;
17    }
18
19    /* allocate the internal buffer, ensure it is on wchar_t boundary */
20    if (! len) {
21        if (new->_data) free(new->_data);
22        new->_data = NULL; new->_len = new->_alloc = 0; new->_flags = 0;
23        new->parent = NULL; new->parts = 0; new->token = NULL;
24        return new;
25    }
26
27    if (new->_data) {
28        if (new->_alloc < len) {
29            s = realloc(new->_data, (len + sizeof(wchar_t)) * sizeof(*s));
30            if (! s) {
31                perror(ERR(string_alloc, realloc));
32                free(new->_data); free(new);
33                return NULL;
34            }
35            new->_data = s; new->_alloc = (len + sizeof(wchar_t)) * sizeof(*s);
36        }
37    } else {
38        new->_data = malloc( (len + sizeof(wchar_t)) * sizeof(*new->_data));
39        if (! new->_data) {
40            perror(ERR(string_alloc, malloc));
41            free(new);
42            return NULL;
43        }
44        new->_alloc = (len + sizeof(wchar_t)) * sizeof(*new->_data);
45    }
46
47    /* copy the given data in the internal buffer */
48    if (string) memmove(new->_data, string, len);
49
50    memset(new->_data + len, 0, sizeof(wchar_t));
51
52    new->_len = len; new->_flags = 0;
53    new->parent = NULL; new->parts = 0; new->token = NULL;
54
55    return new;
56 }

```

Comme on peut le voir, l'allocateur utilisera si possible un bloc mémoire du cache, en utilisant l'allocateur système en dernier recours. L'allocateur suppose que le cache est rempli par le destructeur, dont l'implémentation est donnée ci-après.



```
1 m_string *string_free(m_string *string)
2 {
3     /** @brief clean up a m_string and its internal buffer */
4
5     _m_string_TLS *tls_data = NULL;
6
7     if (! string) return NULL;
8
9     tls_data = pthread_getspecific(TLS);
10
11    string_free_token(string);
12
13    if (string->_flags & _STRING_FLAG_NALLOC) return NULL;
14
15    if (tls_data->free_string < STRING_CACHE_SIZE) {
16        if (~string->_flags & _STRING_FLAG_NOFREE) {
17            string->_data = NULL;
18            string->_len = string->_alloc = 0;
19        }
20        tls_data->string[tls_data->free_string++] = string;
21    } else {
22        if (~string->_flags & _STRING_FLAG_NOFREE)
23            free(string->_data);
24        free(string);
25    }
26
27    return NULL;
28 }
```

Le destructeur, si le cache n'est pas plein, nettoie chaque structure et la prépare à être utilisée plus tard par l'allocateur avant de l'enregistrer en cache.

On voit ainsi qu'à condition d'avoir une utilisation mémoire de la forme «création - destruction - création» et des données de taille relativement constante, il est possible de recycler complètement les blocs mémoire existants et donc d'amortir considérablement le coût de leur allocation initiale.

Bien entendu, dans une application réelle l'utilisation mémoire ne suit que rarement cette forme idéale, mais les gains restent conséquents (voir Annexe A) avec une réduction du nombre d'allocations d'environ 50%.



## A Annexe : Quelques chiffres

La suite d'outils **Valgrind** permet d'évaluer avec précision la quantité d'allocations réalisées par une application. Nous l'avons utilisée ici afin de chiffrer les gains que l'utilisation du cache TLS nous a apporté dans le cadre de notre projet.

Voici le nombre d'allocations (appels à **malloc(3)**, **calloc(3)**, **realloc(3)**) réalisées par notre serveur lors de son démarrage et cinq minutes d'exécution, avec un unique client connecté :

```
malloc/free : 560,687 allocs, 560,249 frees, 10,079,622 bytes allocated.
```

En utilisant le cache d'allocation TLS, le nombre total d'allocations dans les mêmes conditions diminue d'un peu plus de 50% :

```
malloc/free : 202,194 allocs, 201,349 frees, 7,127,472 bytes allocated.
```

On constate cependant que l'occupation memoire totale ne diminue que de 30% ; en effet, le serveur alloue environ deux mégaoctets de ressources variées au démarrage et ces allocations ne peuvent donc pas être optimisées.