TOPPERS 活用アイデア・アプリケーション開発コンテスト

部門　：　活用アイデア部門

作品のタイトル　：　TLSF+TECS
（TECS を用いた動的メモリアロケータのコンポーネント設計）

作成者　：　山本拓朗（大阪大学）

共同作業者　：　大山博司（オークマ株式会社），安積卓也（大阪大学）

対象者　：　TECS を利用するソフトウェア開発者

使用する開発成果物　：　TECS

目的・狙い

以下の目的から TLSF メモリアロケータのコンポーネント化を行った．

- 複数のスレッドが並行動作しても排他制御なしでスレッドセーフに動作する
- 各スレッドで独自のヒープ領域を容易に設定できる
- コンポーネント化により再利用性が向上するため，様々なシステムに拡張できる
- コンポーネント図により可視化できるため，機能役割の理解が容易になる

アイデア/アプリケーションの概要

TLSF メモリアロケータは組込みシステムに適した動的メモリアロケータであるが，現状の問題点として，複数のスレッドが並行動作するとメモリ競合が起きる可能性がある．この問題点を解決するため，TECS を用いて TLSF メモリアロケータのコンポーネント化を行った．コンポーネント化された TLSF メモリアロケータは，各コンポーネントが独自のヒープ領域を保持するため，スレッドセーフに動作することができ，メモリサイズの設定なども容易になる．利用例として，mruby on TECS フレームワークに TLSF コンポーネントを組み込み，マルチ VM 機能を実現した．

はじめに

多くの組込みシステムは，厳しいリソース制約を持っており，効率的なメモリ確保が要求される．さらに，リアルタイム性を要求される組込みシステムでは，メモリ確保の実行時間も重要となる．

このような要求を満たす，組込みシステムに適した動的メモリアロケータとして，TLSF (Two-Level Segregate Fit) アロケータが提案されている．TLSF アロケータは，メモリブロックを 2 段階で細かく分類することでメモリ利用効率を向上させており，常に O(1)で実行するため，最悪実行時間を見積もることができ，リアルタイムシステムに適した動的メモリアロケータである．TLSF アロケータは，メモリ利用効率が良く，リアルタイムシステムに向いているため，多くのリアルタイム OS で採用されている．

しかし，現状では，複数のスレッドが並行動作すると，メモリの衝突が起きる場合がある．本稿では，TECS を用いて TLSF メモリアロケータのコンポーネント化を行った．コンポーネント化された TLSF メモリアロケータは，各コンポーネントが独自のヒープ領域を保持するため，スレッドセーフなメモリアロケータを実現できる．さらに，TECS を用いたコンポーネントベース開発は，ソフトウェアの再利用性を向上させ，システムを可視化できるため，ソフトウェア開発の生産性を向上させることができる．

TLSF について

TLSF (Two-Level Segregate Fit) メモリアロケータは，M. Masmano らによって提案されたリアルタイムシステムに適した動的メモリアロケータである．TLSF メモリアロケータは以下のような特徴がある．

- リアルタイム性

  メモリの確保や解放にかかる最悪実行時間はデータサイズに依存しないため，常に O(1) で実行される．応答時間を見積もることができるため，リアルタイムシステムに適したメモリアロケータである．

- 高速

  最悪実行時間が常に見積もれることに加え，TLSF は高速に実行される．

- 効率的なメモリ消費

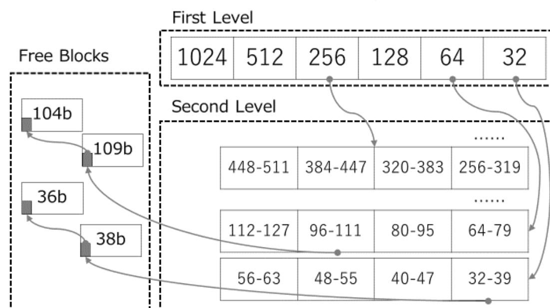
  TLSF では，メモリの断片化(フラグメンテーション)を抑えることで，メモリ効率化を実現している．様々なテストで，平均フラグメンテーション 15 %未満，最大フラグメンテーション 25 %未満を得ている．

TLSF アルゴリズム

TLSF アルゴリズムは，メモリブロックを 2 段階に分類して，要求されたメモリサイズに最適なメモリブロックを検索する．例として，malloc(100) というメモリ確保の要求がさ

れた場合を考える．まず第一段階では，要求されたメモリサイズの最上位ビットで分類する．この場合, 100 を 2 進数で表すと 1100100 であるため, 最上位ビットから 64 から 128 の範囲であることが分かる．次に第二段階では，さらに細かい分類を行う．今回の場合，64 から 128 を 4 分割しており，100 は 96-111 のブロックに入る．この範囲にあるフリーブロックを使用するという流れである．

単純な固定サイズブロック確保では最大 50%の無駄を生じてしまうが, TLSF では 2 段階で細かく分類するため, メモリ効率が良いアルゴリズムとなっている．さらに，検索にかかる時間も高速で常に同じ速度で実行される．



コンポーネント設計

アロケータが使用するメモリ管理用のシグニチャ記述を以下の通り．メモリプール初期化関数 initializeMem-oryPool，メモリ確保用の関数 calloc, malloc, realloc，そしてメモリ解放用の関数 free をシグニチャとして定義している．

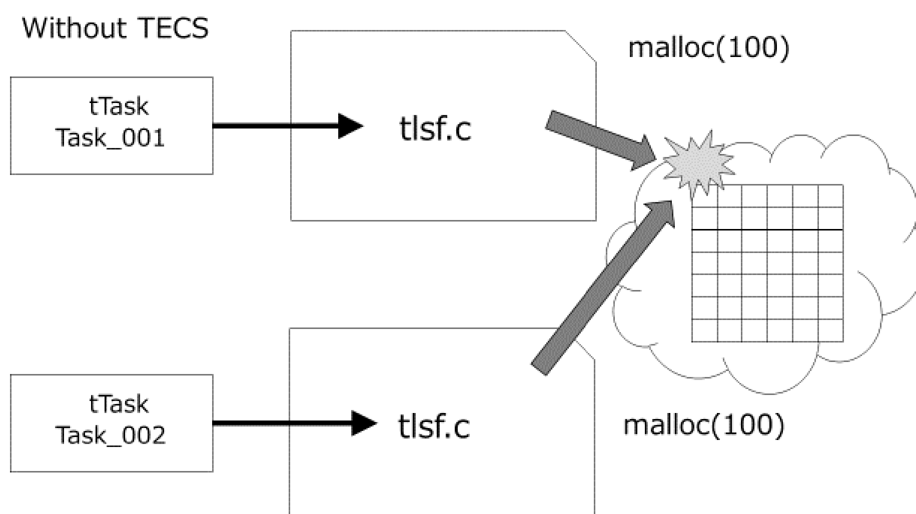
```
1 signature sMalloc {
2     int initializeMemoryPool(void);
3     void *calloc( [in]size_t nelem,
4                   [in]size_t elem_size );
5     void *malloc( [in]size_t size );
6     void *realloc( [in]const void *ptr,
7                    [in]size_t new_size );
8     void free( [in]const void *ptr );
9 };
```

次に TLSF メモリアロケータコンポーネントのセルタイプ記述を示す．受け口 eMalloc は，メモリ管理(確保や解放) を行うすべてのコンポーネントと結合される．ここで，[inline] は受け口関数をインライン関数として提供するための指定子である．メモリプールサイズをコンポーネントの属性として，メモリプールへのポインタをコンポーネントの内部変数として定義している．各コンポーネント

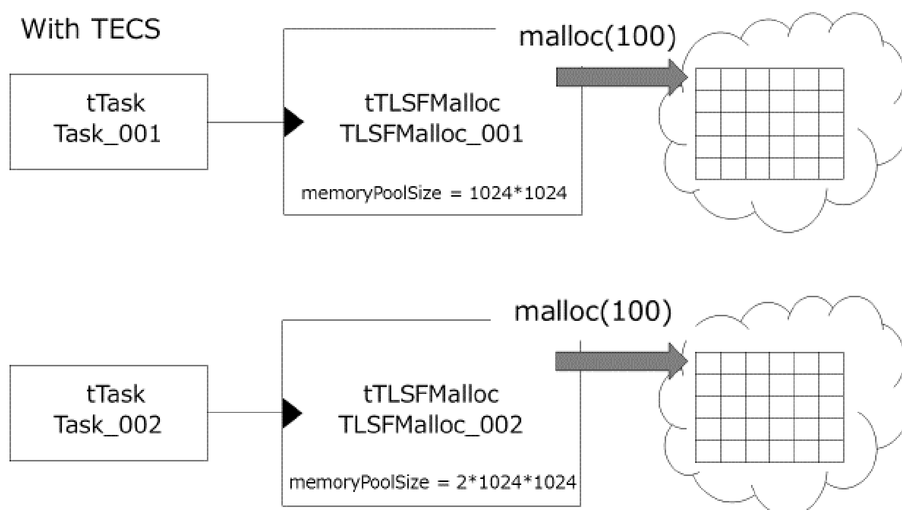
```
1  celltype tTLSFMalloc {
2      [inline]
3          entry sMalloc eMalloc;
4      attr {
5          /* memory pool size in bytes */
6          size_t memoryPoolSize;
7      };
8      var {
9          [size_is( memoryPoolSize / 8 )]
10             uint64_t *pool;
11     };
12 };
```

が独自のヒープ領域を保持しているため，異なるスレッドで同時にメモリ管理用の関数を呼び出した場合でも，排他制御を行うことなく，メモリ競合を起こさずに動作させることが可能になる．

下図 に示すように，コンポーネント化を行う前のTLSFでは，ヒープ領域を複数のスレッドで共有しているため，複数のスレッドからメモリの確保や解放を同時に行うとメモリ競合が生じる場合があった．



次のように，TECSを用いてTLSF のコンポーネント化を行うと，各コンポーネントが独自にヒープ領域を保持し，その中でメモリ管理を行うため，排他制御なしでスレッドセーフに動作することが可能になる．



上のコンポーネント図で表される TLSF メモリアロケータコンポーネントの組上げ記述を示す．2 組のタスクコンポーネントと TLSF コンポーネントが結合されている．それぞれのメモリプールサイズは 5 行目及び 11 行目のように内部変数として設定可能である．

```
1  cell tTask Task_001 {
2      cMalloc = TLSFMalloc_001.eMalloc;
3  };
4  cell tTLSFMalloc TLSFMalloc_001 {
5      memoryPoolSize = 1024*1024; /* 1MB */
6  };
7  cell tTask Task_002 {
8      cMalloc = TLSFMalloc_002.eMalloc;
9  };
10 cell tTLSFMalloc TLSFMalloc_002 {
11     memoryPoolSize = 2*1024*1024; /* 2MB */
12 };
```

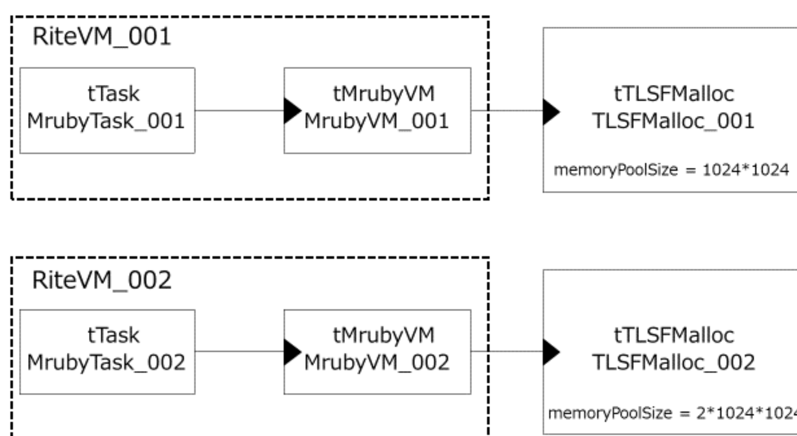
TLSF メモリアロケータコンポーネントの受け口関数を実際に呼び出しているコード部分を以下に示す．利用部分は， mruby on TECS フレームワークで，VM がメモリ管理を行う関数である．8 行目は，TLSF メモリアロケータコンポーネントの free 関数を呼び出している．cMalloc は，呼び口名を表している.同様に，13 行目，17 行目はメモリ確保を行っている部分である．コードが実行されるセルが，Task 001 の場合は TLSFMalloc_001 コンポーネントで，Task 002 の場合は TLSFMalloc_002 コンポーネントでそれぞれメモリ管理が行われる．このように，TECS を用いたコンポーネントベース開発では，それぞれ結合しているセルは異なるものの，実際に C コードを修正することなく，同じコードで動作させることが可能になる．

```
1  void*
2  mrb_TECS_allocf(mrb_state *mrb, void *p,
3                  size_t size, void *ud)
4  {
5      CELLCB *p_cellcb = (CELLCB *)ud;
6      if (size == 0) {
7          //tlsf_free(p);
8          cMalloc_free(p);
9          return NULL;
10     }
11     else if (p) {
12         //return tlsf_realloc(p, size);
13         return cMalloc_realloc(p, size);
14     }
15     else {
16         //return tlsf_malloc(size);
17         return cMalloc_malloc(size);
18     }
19 }
```

mruby on TECS での利用例

mruby on TECS は，mruby (軽量 Ruby)を用いたコンポーネントベース開発が可能な組込みソフトウェア開発フレームワークである．スクリプト言語を用いることで，組込みソフトウェア開発の生産性向上を目指している．一般に，スクリプト言語は実行速度が遅いため，組込みソフトウェアに向いていないが，mruby on TECS では，mruby-TECS ブリッジの機能によって, mruby プログラムから C 言語の関数を呼び出すことができるため，C 言語と変わらない速度でプログラムを実行することができる．mruby プログラムは，mruby コンパイラによってバイトコードに変換され，RiteVM と呼ばれる VM 上で実行される．本フレームワークでは，RiteVM やリアルタイム OS の機能もすべて TECS によってコンポーネント化されている．

mruby on TECS フレームワークでは, VM が行うメモリ管理に TLSF メモリアロケータを採用している．しかし，既存の TLSF メモリアロケータではスレッドセーフではないため，複数のスレッドからメモリの確保や解放を行うと，メモリ衝突が起きてしまう．VM は高い頻度でメモリの確保と解放を繰り返すため，マルチ VM として複数の VM を起動すると，すぐに衝突を起こしてしまう．そこで下図のように，VM に TLSF メモリアロケータコンポーネントを結合し，各 VM で独自のヒープ領域を持つように設計する．各 VM が結合している TLSF コンポーネントがそれぞれメモリプールを保持しているため，複数の VM がメモリ衝突を起こすことなく，実行できる．

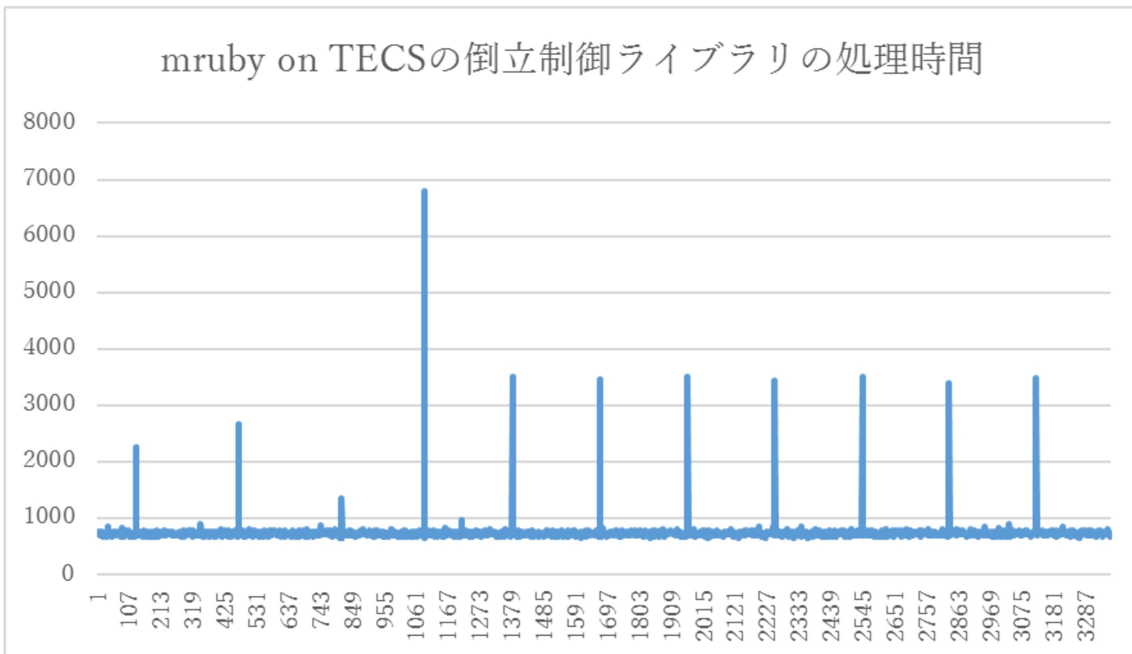


1つ目の VM が 1MB (1024*1024) のヒープ領域, 2つ目の VM が 2MB (2*1024*1024) のヒープ領域を持っている．コンポーネント化により，各 VM で異なるヒープ領域を設定することが容易になっている. さらに, RiteVM はインクリメンタル GC (Garbage Collection) を行うが，独自のメモリプールを保持しているため，GC を始めた VM が GC の実行によって，他の VM の実行を妨げることもない．

今後の課題

提案する TLSF コンポーネントの拡張として，動的メモリの利用状況の統計情報を取得できる機能を提供する．メモリ確保の頻度や残りのヒープ領域サイズ等の情報を取得することで，mrubyVM の GC の動作状況を分析することができる．

例えば mruby on TECS では，ある一定周期で処理時間がもたつくことがある（下図）．これは VM の GC が実行されているのが原因と考えられる．このような VM の動作状況を分析することに役立つと考えられる．



統計情報を取得するためのコンポーネントは，以下のようなコンポーネント設計を考えている．このコンポーネントは，TECS の複合セルまたは Through plugin を用いて実装することで，容易に取り外しを可能にする．