

MemoryManager

目次

1. 概要	3
2. 関数リファレンス.....	3
InitializeMemoryManager	4
TerminateMemoryManager.....	5
AllocateMemory.....	6
FreeMemory.....	7
3. 仕組み	8
更新履歴	11

1. 概要

メモリの分譲・空きメモリの統合等、メモリ管理を行うモジュールである。

2. 関数リファレンス

本モジュールには、Table2-1.の関数が含まれる。

Table2-1. 関数一覧

関数名	概要
InitializeMemoryManager	管理するメモリマネージャの初期化
TerminateMemoryManager	メモリマネージャの後始末
AllocateMemory	メモリブロックの確保
FreeMemory	メモリブロックの解放

InitializeMemoryManager

書式 :

```
void InitializeMemoryManager( MEMORY_MANAGE_T *pManage,  
                             void *pStart, void *pEnd );
```

機能 :

メモリマネージャの初期化

引数 :

pManage ... 初期化するメモリマネージャのインスタンス

pStart ... 管理するメモリの開始アドレス

pEnd ... 管理するメモリの終了アドレス

返値 :

なし

詳細 :

pManage で示されるメモリマネージャ構造体を初期化して、メモリマネージャとして機能するようにする。本関数で初期化していない MEMORY_MANAGE_T 構造体を使って、本関数以外の本モジュールの関数を呼びだしてはならない。

pStart, pEnd が示すアドレスは管理対象に含まれる。

pManage の内容を利用者が書き換えてはならない。MEMORY_MANAGE_T の内容は将来予告無く変更される可能性があるため、デバッグ用途以外で読み取りも行うべきでない。

参考 :

[TerminateMemoryManager](#)

TerminateMemoryManager

書式 :

```
void TerminateMemoryManager( MEMORY_MANAGE_T *pManage );
```

機能 :

メモリマネージャの後始末

引数 :

pManage ... 初期化済みのメモリマネージャのインスタンス

返値 :

なし

詳細 :

初期化時に指定されたメモリ領域を、メモリマネージャの管理下から除外する。
本関数により後始末された MEMORY_MANAGE_T 構造体は、未初期化の状態に戻る。

参考 :

[InitializeMemoryManager](#)

AllocateMemory

書式 :

```
void *AllocateMemory( MEMORY_MANAGE_T *pManage, UINT32_T Size );
```

機能 :

メモリブロックの確保

引数 :

pManage ... 初期化済みのメモリマネージャインスタンス

Size ... 確保したいメモリのバイト数

返値 :

NULL ... 失敗。指定のバイト数を確保できなかった。

!NULL ... 成功。確保したメモリブロックの先頭アドレス。

詳細 :

pManage で示されるメモリマネージャの管理下にあるメモリから、指定のサイズのメモリブロックを分譲してもらう。

得られたメモリブロックは、Size で指定されたサイズが連続的に確保されていることが保証され、FreeMemory で解放するまで利用できる。

不要になったメモリブロックは FreeMemory で解放することにより、再利用されるようになる。

Size に 0 を指定すると必ず失敗する。

参考 :

[FreeMemory](#)

FreeMemory

書式 :

```
void FreeMemory( MEMORY_MANAGE_T *pMangage, void *pMemory );
```

機能 :

確保したメモリを解放する

引数 :

pManage ... 初期化済みのメモリマネージャインスタンス

pMemory ... 確保済みのメモリ

返値 :

なし

詳細 :

pMemory に NULL を指定すると何もせずに抜ける。

pMemory は pManage の管理下にあるメモリでなければならない。異なる MEMORY_MANAGE_T インスタンスの管理下にある pMemory を指定した場合の動作は保証されないため注意すること。

参考 :

[AllocateMemory](#)

3. 仕組み

メモリマネージャは、2つの双方向リストを持ち、一方は確保済みブロックの一覧、もう一方は未使用ブロックの一覧を示す。メモリマネージャが管理すべきメモリ領域は、複数のメモリブロックの集合として扱われる。メモリブロックは、この双方向リストのリンク情報と、ブロック自身のサイズ、シグネチャの情報を持つ管理情報を持っている。このイメージを Fig3-1.に示す。

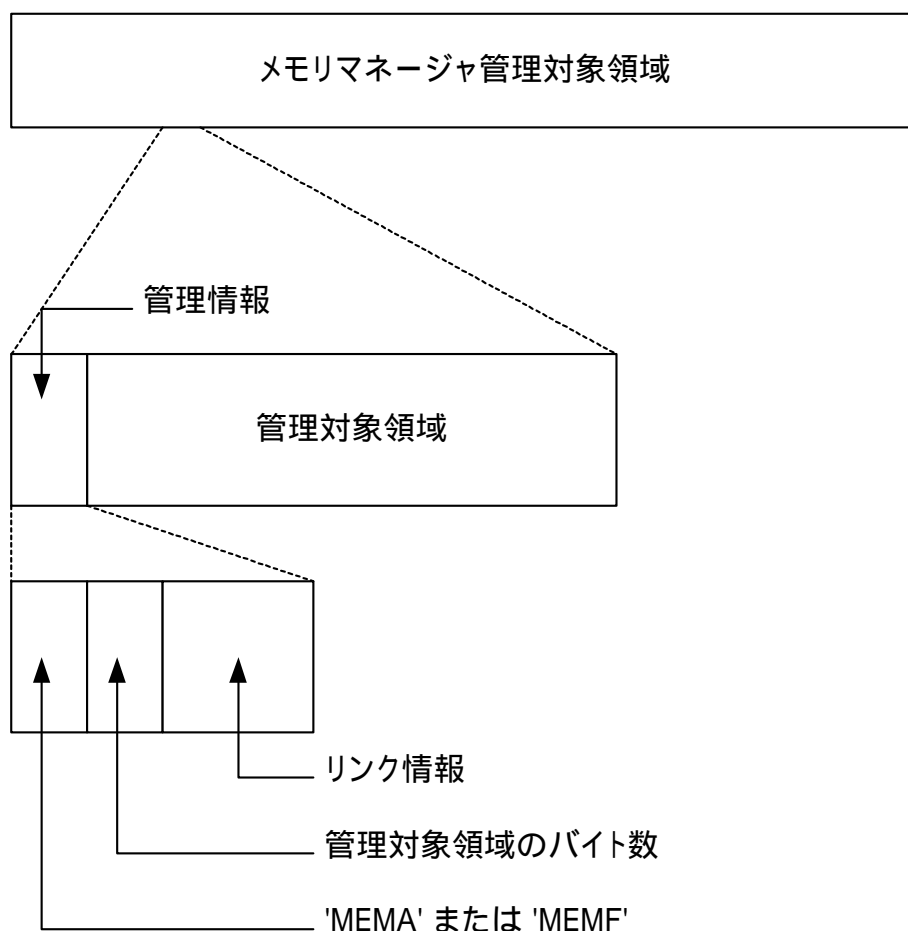


Fig3-1. メモリの管理イメージ

初期化時は、メモリマネージャ管理対象領域内にただ一つのメモリブロックが存在し、Fig3-2 のように未使用領域リストに接続されている。

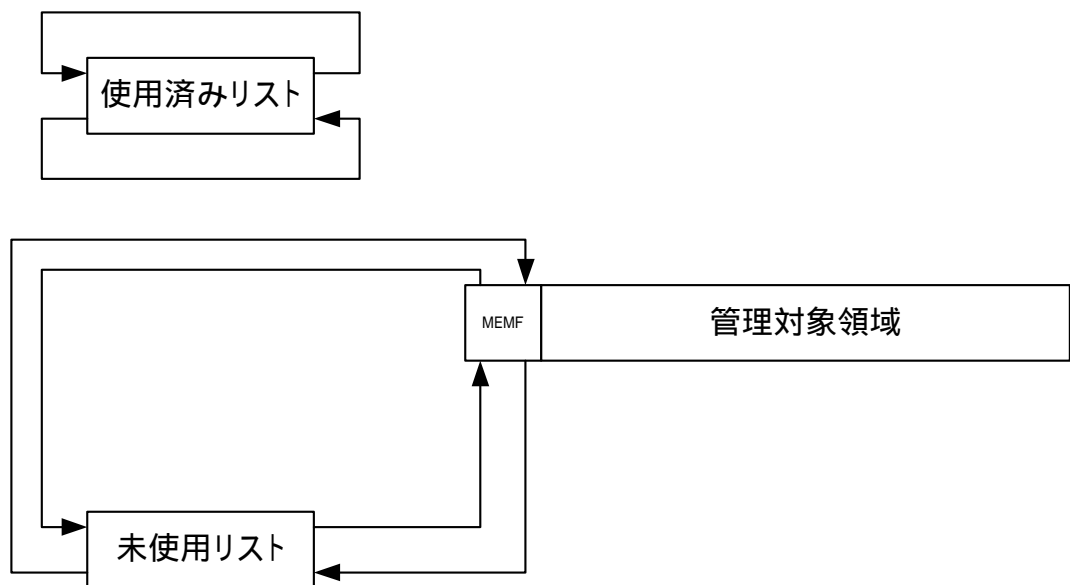


Fig3-2. 初期化直後の状態

AllocateMemory が呼ばれると、未使用リストの先頭から順に要求サイズ以上のブロックを検索する。このとき FirstFit 法を採用するため、未使用リストはアドレス順にソートしておくものとする。AllocateMemory の度にソートするのではなく、未使用リストに追加するブロックを挿入ソート法で挿入することで、ソートの負担を最小限に抑える。この挿入ソートを実現するために、シグネチャを使う。

使用済みリストに接続されているブロックのシグネチャは 'MEMA'、未使用リストに接続されているブロックのシグネチャは 'MEMF' となるように更新する。メモリマネージャで管理される領域は連続メモリであるため、未使用リストへ追加するブロック = FreeMemory されたブロックの管理領域の次のアドレスから、Fig3-3 のようにそのブロックのサイズを加算したアドレスが、隣接する次のブロックの管理領域となる。

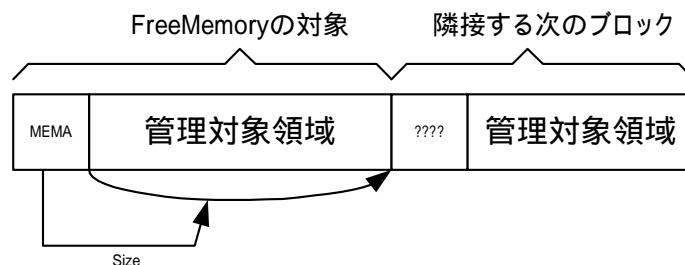


Fig 3-3. 隣接する次のブロックのアドレス計算

隣接する次のブロックのシグネチャを読み取り、'MEMA' なら使用済みブロック。'MEMF'なら未使用ブロックであることがわかる。もし使用済みブロックなら、未

使用ブロックが見つかるまで同じことを繰り返す。もし未使用ブロックなら、その管理領域は未使用ブロックリストに接続されているため、そのブロックの直前に挿入することでアドレス順にソートされた状態の未使用リストを得られる。

検索を続け、未使用ブロックが最後まで見つからなかった場合は、未使用リストの最後に追加する形で挿入すれば目的を達成できる。

更新履歴

更新日	リビジョン	更新内容	更新者
2004/07/19	1.0.0	初版	t.hara