CodeZine
開発者のための実装系Webマガジン

特集記事

50年前に作られたメモリ管理アルゴリズム「Buddy memory allocation」

C C++

WEB用を表示

επιστημη[著]

2016/04/20 14:00

ダウンロード↓ サンプルファイル (16.3 KB)

4月ですねー、この世界に身を置いて三十数年目の春です。当時のノートを読み返すと、組み込み用途のメモリ管理ルーチンをアセンブラで書いたりしてました。黄ばんだノートの覚え書きの中から、メモリ確保/解放のスピードを重視したメモリ管理アルゴリズム:Buddy memory allocationを紹介しましょうか。

Buddy memory allocationの仕組み

Buddy memory allocationは、僕がまだ駆け出しのプログラマだった30年以上前に先輩に教わったもので、かなり名のあるアルゴリズムかと思っていたのですが、日本語版Wikipediaには載っていないみたいで、**本家英語版で見つけました**。それによりますとBuddy memory allocationが考案されたのは1963年とのこと、50年以上前に作られたアルゴリズムなんですね。

Buddy memory allocationが管理する対象は2^L個の連続したメモリ・ブロック。ブロックの大きさは任意で、これを最小単位として確保/解放が行われます。ブロック数2^LのLをlevelまたはorderと呼び、orderが10の場合2^10=1024個の連続したメモリ・ブロックを扱うことになります。1blockが1KBなら1MBのメモリ・プールです。

Buddy memory allocationが行うのはメモリ・ブロックの確保と解放すなわち:

- **確保**:プログラムから要求されたブロック数をnとするとき、このnより小さくない最小の2^x個の未使用連続ブロックを確保してプログラムに返すこと
- **解放**:プログラムが使用を終えたブロックを未使用状態とし、以降の確保に備えること

確保時のふるまいが特徴的です。確保するブロック数が2のベキになっています。場合によっては無駄に大きな領域が確保されることがありますが、この"縛り"のおかげで確保/解放に要する時間を小さく抑えることができます。そのカラクリを解説します。

order-4, 64KB/blockのBuddy memory allocationで説明しましょう。order-4ですからblock数は2^4=16、64KB/blockなら扱うメモリの総量は16*64=1024KBです。

(1)まず初期状態:order-4(16個)のblockを未使用状態にしておきます(□は未使用block)。

1: □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ 初期状態

(2)プログラムAが34KBのメモリ確保を要求しました。34KBなら必要なblock数は1、すなわちorder-0のblockです。この時点で管理領域にあるのはorder-4のblock1つ。これでは要求される量より大きすぎるので、これを半分に切り分けorder-3のblock2つに分割します。2つに分割されたblockの組をbuddy(相方/相棒)

と呼ぶことにします。

2.1: □ □ □ □ □ □ □ □|□ □ □ □ □ □ □ □ 分割

order-3ではまだ大きい。左のblockをさらに2分割。

2.2: □ □ □ □|□ □ □ □|□ □ □ □ □ □ □ □ 分割

さらにもう一回。

2.3: □ □|□ □|□ □ □ □|□ □ □ □ □ □ □ □ 分割

まだ大きいのでもう一回。

2.4 : □|□|□ □|□ □ □ □|□ □ □ □ □ □ □ □ 分割

これでようやく要求されたサイズを満たす最小のblockができました。左端のblockを使用中(■)とし、その位置(=0)をプログラムAに返します。

2.5: ■|□|□ □|□ □ □ □|□ □ □ □ □ □ □ □ 確保(0)

(3)プログラムBが66KBの確保を要求してきました。64KB/blockですから必要なblock数は2、order-1のblockです。管理領域にはorder-1の未使用blockがあるので、これを使用中としプログラムBに返します。

3: ■|□|■ ■|□ □ □ □|□ □ □ □ □ □ □ □ 確保(2-3)

(4)さらにプログラムCが35KBの確保を要求。block数1、つまりorder-0ですね。プログラムAに貸し出したblockのbuddy(相方)が未使用なのでこれを返しましょう。

4: ■|■|■ ■|□ □ □ □|□ □ □ □ □ □ □ □ 確保(1)

(5)プログラムDが67KBを要求します。order-1のblockです。order-1のblockは全部使用中で空きがありません。未使用のorder-2のblockを分割しorder-1を2つ作ります。

5.1: ■|■|■ ■|□ □|□ □|□ □ □ □ □ □ □ □ 分割

order-1のblockが2つできたので、その一つをプログラムDに返します。

5.2: ■|■|■ ■|■ ■|□ □|□ □ □ □ □ □ □ □ 確保(4-5)

(6)プログラムBが(3)で確保したメモリを開放します。order-1のblockが1つ空きました。

6: ■|■|□ □|■ ■|□ □|□ □ □ □ □ □ □ □ 解放(2-3)

(7)プログラムDが(5)で確保したメモリを返します。order-1のblockがもう一つ空きました。

7.1: ■|■|□ □|□ □|□ □|□ □ □ □ □ □ □ □ 解放(4-5)

すると、お互いがbuddyの関係にある2つのorder-1 blockが空いたので、これを1つにまとめてorder-2にします

7.2: ■|■|□ □|□ □ □ □|□ □ □ □ □ □ □ □ 結合(4-7)

(8)プログラムAが(2)で確保したメモリを解放します。

8: □|■|□ □|□ □ □ □|□ □ □ □ □ □ □ □ 解放(0)

(9)プログラムCが(4)で確保したメモリを開放します。

9.1 : □|□|□ □|□ □ □ □|□ □ □ □ □ □ □ □ 解放(1)

order-0の空きが2つあるのでまとめます

9.2: □ □|□ □|□ □ □ □|□ □ □ □ □ □ □ □ 結合(0-1)

それによってorder-1の空きが2つできました。まとめます。

9.3: □ □ □ □|□ □ □ □|□ □ □ □ □ □ □ □ 結合(0-3)

もう一度。

9.4 : □ □ □ □ □ □ □ □|□ □ □ □ □ □ □ □ 結合(0-7)

さらにもう一度。これで最初の状態に戻りました。

9.5: □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ 結合(0-15)

……こんなダンドリ。半分ずつに切り分け(orderを下げる)て確保し、解放時はできるだけまとめる(orderを上げる)ことを繰り返します。一度の分割/結合でblockの大きさが半分/二倍になるのですから、確保/解放に要する時間はlogNのオーダーとなりますね。

未使用領域を線型リストで管理する方法では未使用blockが数珠つなぎになるために、空き領域の検索に要する時間は分断された未使用blockの数Nに比例しますし、小さなblockの確保/解放がランダムに繰り返されると未使用blockのところどころに歯抜け(fragmenentation:分断化)が起こりやすくなります。Buddy memoru allocationだと2のベキに切り分けることで分断化が起こりにくくなります。

Buddy memory allocationは確保/解放のスピードが大きな利点ですが、最大の欠点はorderが大きなblockの確保時に無駄が大きくなりがちなことです。確保するblock数が2のベキであれば問題ないけど、例えば9blockの確保要求に対しては16block確保するので、ほぼ半分の領域が無駄になっちゃいます。

Buddy memory allocationの実装

30年ほど前こいつを8086アセンブラで書きました。C/C++で実装されたものがないか探してみたらば**GitHubに見つけましたよ**。ここにあったC実装、C99でサポートされたinlineを使っているためコンパイルエラーとなる処理系もありますが、inlineを潰してしまうかC++としてコンパイルすれば無問題です。C++用にほんの少し修正を加えたもの(buddy.h, buddy.cpp)を**サンプル**に同梱しておきました。

インターフェースを軽く説明しておきます:

struct buddy* buddy_new(int lebel)

buddyを生成します。引数levelはbuddyが管理するblockの初期orderで、例えば buddy_new(10) で2^10=1024個のblockを管理します。

void buddy_delete(struct buddy*)

buddyの廃棄。生成時に作業領域をmallocしているので、用が済んだらbuddy_deleteをお忘れなく。

int buddy_alloc(struct buddy*. int size)

少なくともsize個の連続blockを確保し、確保された連続blockの先頭block番号を(0起点で)返します。確保できなかったら-1です。

int buddy_free(struct buddy*. int offset)

buddy_allocで確保されたblockを解放します。

int buddy_size(struct buddy*, int offset)

buddy_allocで確保された実際のblock数を返します。

buddy_dump(struct buddy*)

blockの使用状況をstdoutに出力します。block番号bとその大きさsが、未使用なら(b:s)確保済なら[b:s]と表現されます。

前述のダンドリ説明(1〜9)を実装/実行したのがコチラ:

list-1

```
#include <stdio.h>
#include "buddy.h"

int test_alloc(struct buddy* b, int sz) {
  int r = buddy_alloc(b, sz);
  printf("alloc %d (sz= %d)\n",r,sz);
  buddy_dump(b);
  return r;
}

void test_free(struct buddy* b, int addr) {
  printf("free %d\n",addr);
  buddy_free(b, addr);
  buddy_dump(b);
}

int main() {
  struct buddy* b = buddy_new(4);

  printf("--- 1. initial ----------------:\n");
  buddy_dump(b);
  printf("--- 2. A allocates 1 block ----: ");
  int mA = test_alloc(b, 1);
  printf("--- 3. B allocates 2 blocks ---: ");
  int mB = test_alloc(b, 2);
  printf("--- 4. C allocates 1 block ----: ");
  int mC = test_alloc(b, 1);
  printf("--- 5. D allocates 2 blocks ---: ");
  int mD = test_alloc(b, 2);
  printf("--- 6. B frees ----------------: ");
  test_free(b,mB);
  printf("--- 7. D frees ----------------: ");
  test_free(b,mD);
  printf("--- 8. A frees ----------------: ");
  test_free(b,mA);
  printf("--- 9. C frees ----------------: ");
  test_free(b,mC);
  printf("\n");

  buddy_delete(b);
}
```

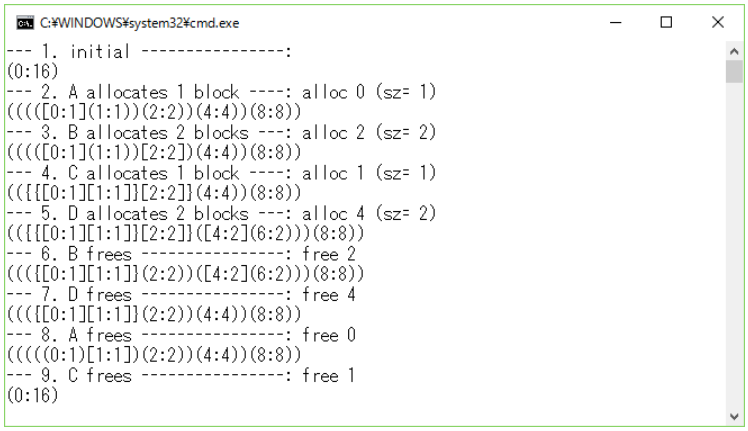

fig-1

……ちょい待ち、このコード、メモリの確保も解放もやってないやん? そのとおり、buddyそれ自体は連続する2^L個のblockそれぞれの使用/未使用状態を管理するだけなんです。

これは実用にはならんので、確保時にはポインタを返し解放時にはポインタを渡すよう、簡単なwrapper:buddy_poolを用意しました。

list-2

```
#ifndef BUDDY_POOL_H_
#define BUDDY_POOL_H_

#include "buddy.h"
#include <mutex>
#include <cstdint>
#include <cstddef>

class buddy_pool {
private:
  std::mutex    mutex_;  // mutex
  buddy*        buddy_;  // Buddy
  std::uint8_t* buffer_; // memory pool
  std::size_t   block_;  // bytes per block

public:
  buddy_pool(int level, std::size_t block, void* pool)
    : buffer_(static_cast<uint8_t*>(pool)), block_(block) {
    buddy_ = buddy_new(level);
  }

  ~buddy_pool() {
    buddy_delete(buddy_);
  }

  void* allocate(std::size_t n) {
    std::lock_guard<std::mutex> guard(mutex_);
    int nblock = static_cast<int>((n + block_ -1U)/block_);
    int offset = buddy_alloc(buddy_, nblock);
    return offset < 0 ? nullptr : buffer_ + offset * block_;
  }

  void deallocate(void* ptr) {
    std::lock_guard<std::mutex> guard(mutex_);
    int offset = static_cast<int>((static_cast<uint8_t*>(ptr) - buffer_)/block_);
    buddy_free(buddy_, offset);
  }
```

```
  void dump(void (*func)(const char*)) const {
    buddy_dump_f(buddy_, func);
  }

  void dump() const {
    buddy_dump(buddy_);
  }

  static std::size_t required(int level, std::size_t block) {
    return block << level;
  }

  static buddy_pool* make(int level, std::size_t block, void* pool) {
    return new buddy_pool(level, block, pool);
  }

};
#endif
```

buddy_poolのコンストラクタにはlevel、blockあたりのbyte数、そしてあらかじめ確保された(少なくともblock*(2^level)byteの大きさを持つ)領域を与えます。buddy_poolはこの領域を小分けにして確保します。使い方はmalloc/freeとおんなじですね。

list-3

```
#include "buddy_pool.h"
#include <cstring>
#include <cstdio>

int main() {
  char buffer[1024];
  buddy_pool pool(4, 8, buffer);
  char* hello = (char*)pool.allocate(6);
  char* world = (char*)pool.allocate(6);
  pool.dump();
  strcpy(hello, "Hello");
  strcpy(world, "world");
  printf("%s, %s\n", hello, world);
  pool.deallocate(hello);
  pool.deallocate(world);
}
```

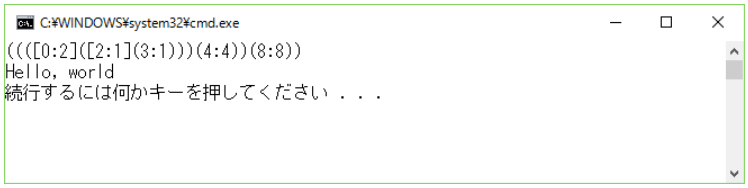

fig-2

おまけにもうひとひねり、std::vectorやstd::listなど標準C++コンテナへのメモリ供給源としてbuddy_poolを利用するallocator:buddy_allocatorを実装します。

list-4

```
#ifndef BUDDY_POOL_H_
#define BUDDY_POOL_H_

#include "buddy.h"
#include <mutex>
```

```
#include <cstdint>
#include <cstddef>

class buddy_pool {
private:
  std::mutex    mutex_;  // mutex
  buddy*        buddy_;  // Buddy
  std::uint8_t* buffer_; // memory pool
  std::size_t   block_;  // bytes per block

public:
  buddy_pool(int level, std::size_t block, void* pool)
    : buffer_(static_cast<uint8_t*>(pool)), block_(block) {
    buddy_ = buddy_new(level);
  }

  ~buddy_pool() {
    buddy_delete(buddy_);
  }

  void* allocate(std::size_t n) {
    std::lock_guard<std::mutex> guard(mutex_);
    int nblock = static_cast<int>((n + block_ -1U)/block_);
    int offset = buddy_alloc(buddy_, nblock);
    return offset < 0 ? nullptr : buffer_ + offset * block_;
  }

  void deallocate(void* ptr) {
    std::lock_guard<std::mutex> guard(mutex_);
    int offset = static_cast<int>((static_cast<uint8_t*>(ptr) - buffer_)/block_);
    buddy_free(buddy_, offset);
  }

  void dump(void (*func)(const char*)) const {
    buddy_dump_f(buddy_, func);
  }

  void dump() const {
    buddy_dump(buddy_);
  }

  static std::size_t required(int level, std::size_t block) {
    return block << level;
  }

  static buddy_pool* make(int level, std::size_t block, void* pool) {
    return new buddy_pool(level, block, pool);
  }

};

#endif
```

おためしはコチラ。

list-5

```
#include "buddy_allocator.h"
#include <cstdio>
```

```
#include <vector>
#include <list>

int main() {
  const size_t block = sizeof(int);
  const int level = 8;
  char buffer[block << level];
  buddy_pool pool(level, block, &buffer);

  buddy_allocator<int> alloc(&pool);
  {
    std::vector<int,buddy_allocator<int>> v(alloc);
    std::list<int,buddy_allocator<int>> l(alloc);
    for ( int i = 0; i < 10; ++i ) {
      v.push_back(i);
      l.push_back(i);
      pool.dump();
      printf("\n");
    }
    for ( int i = 0; i < 5; ++i ) {
      l.pop_front();
      pool.dump();
      printf("\n");
    }
    printf("\n vector : ");
    for ( int item : v ) printf("%3d", item);
    printf("\n list   : ");
    for ( int item : l ) printf("%3d", item);
  }
  printf("\n");
  pool.dump();
}
```

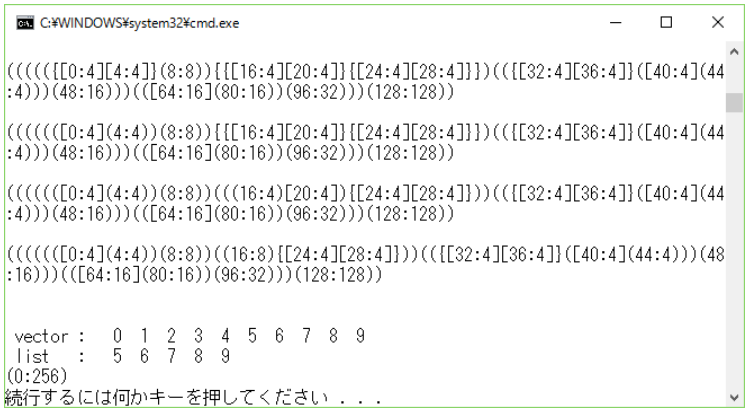

fig-3

30年前の黄ばんだノートから、Buddy memory allocationのご紹介でした。IoTが賑やかになってきたので、芸の肥やしにと遅まきながらRaspberry Piを手に入れたんですよ。組み込み向けの小さなボードでメモリを高速にやりくりするのに使えるんじゃないかと、古ぼけたコードを引っ張り出してホコリを払ってみた次第です。

バックナンバー WEB用を表示

著者プロフィール

επιστημη(エピステーメー)

C++に首まで浸かったプログラマ。Microsoft MVP, Visual C++ (2004.01〜) だったり わんくま同盟でたまにセッションスピーカやったり 中国茶淹れてにわか茶人を気取ってたり、あと Facebook とか。著書: - STL標準講座 (監修) -...

※プロフィールは、執筆時点、または直近の記事の寄稿時点での内容です

ページトップへ

ヘルプ
広告掲載のご案内
著作権・リンク
免責事項
会社概要

スタッフ募集!
メンバー情報管理
メールバックナンバー
マーケティング
エンタープライズ

IT人材
教育ICT
マネー・投資
ネット通販
イノベーション
ホワイトペーパー

プロジェクトマネジメント
書籍・ソフトを買う
電験3種対策講座
電験3種ネット
第二種電気工事士

メンバーメニュー | ログアウト