オペレーティングシステムA特別課題レポート

1W173092-0 富岡祐介

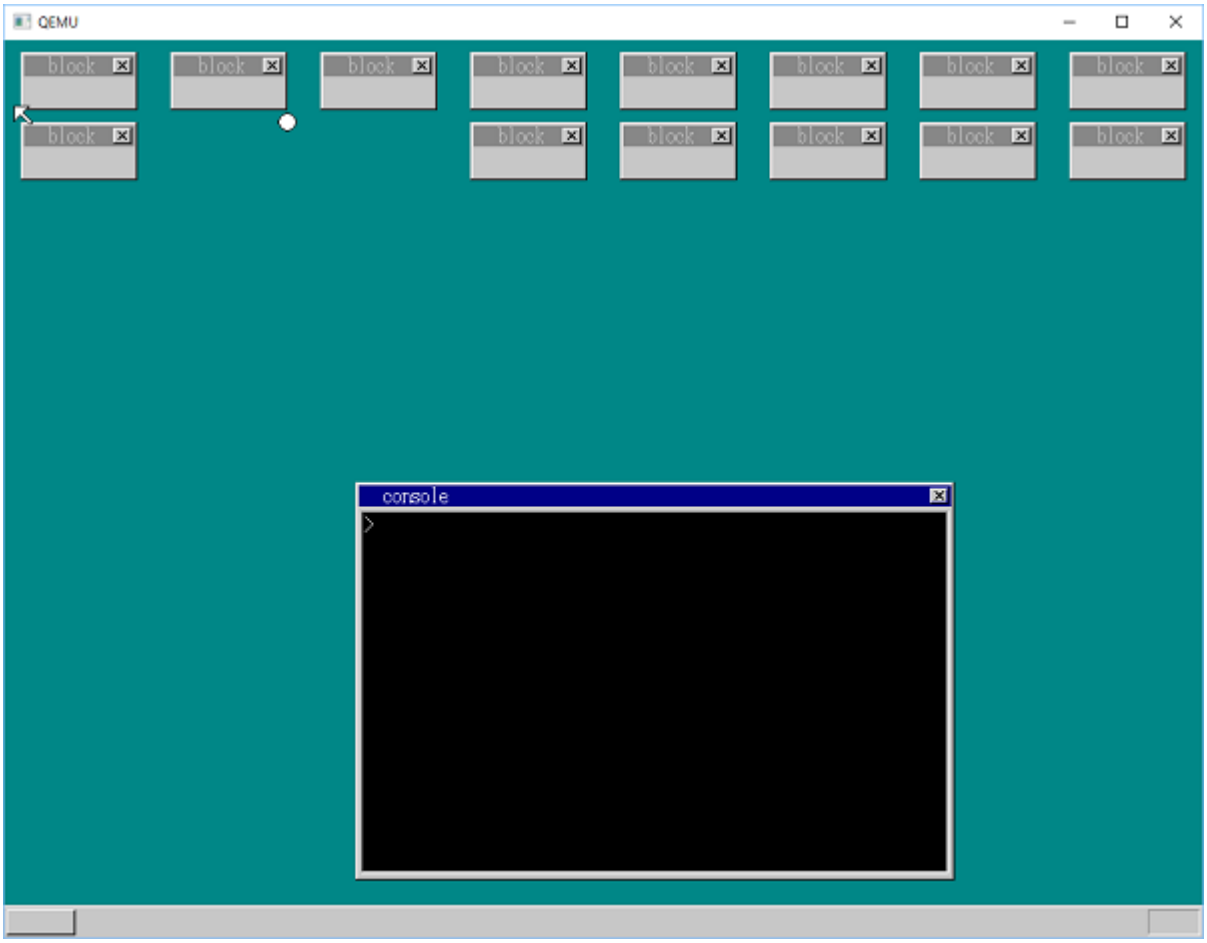
制作OS概観

以下に概要を記述する二つのOSは、オペレーティングシステムAの特別課題に向けて制作されたものである。このレポート内で解説するプロセス管理、メモリ管理、ファイル管理、デバイス管理についてはこれらの二つのOSで実際に用いられているソースコードを含め解説する。

ソースコードはそれぞれ以下のURLにて公開している。

- hariballOS : https://github.com/takowasaby/hariball
- kaOS : https://github.com/takowasaby/kaOS

hariballOS（hariboteOS）



hariballOSは「30日でできる！　OS自作入門」[1]に沿って作成されるOSである「はりぼてOS」にブロック崩しの機能を付け加えたものである。

プロセス管理やメモリ管理、ファイル管理などははりぼてOSとほぼ同一のものとなっているが、特徴として、アプリケーションのレイヤではなく、カーネルのレイヤで実装したブロック崩しが挙げられる。

上に示した画像の通り、一般的なブロック崩しにおけるブロックはウインドウであり、弾や自分で動かすブロックもウインドウとなっている。

ウインドウははりぼてOS上の下敷き（Sheet）の上で描画されており、画素ごとに描画する下敷きの番号を記憶しておくことでウインドウを更新するときに再描画される範囲を最低限に保っている。この性質を弾の当たり判定に利用している。

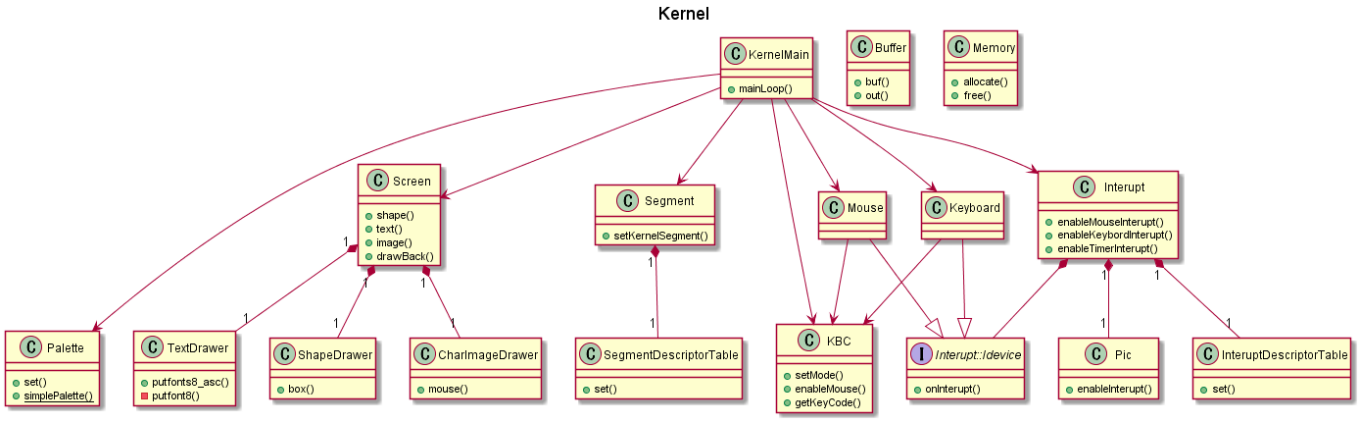
```
// hariball/haribote/ball.c
int up_point = get_map_val(shtctl->map, scrnx, center_point_x, ball->y);
if (up_point != ball->sht - ball->sht->ctl->sheets0 && up_point != 0 && !
(down_point != ball->sht - ball->sht->ctl->sheets0 && down_point != 0))
{
    if ((shtctl->sheets0[up_point].flags & 0x80) != 0)
    {
        sheet_updown(&shtctl->sheets0[up_point], -1);
        ret = 1;
    }
    ball->diry = 1;
}
```

上記のコードは、ボールが進む方向を更新する処理の一部である。`shtctl`と呼ばれる下敷きを管理する構造体から画素ごとの下敷きの番号を受け取り、弾の上端を描画する画素の下敷き番号を導き出している。その後、その値をもとに弾の進む方向を反転させるかどうかを判定している。また、ぶつかったウインドウが消すことのできるブロックにあたるものかどうかを判定し、そうであった場合はウインドウの表示を消している。

このように、カーネルのレイヤで実装されるウインドウの仕組みを利用してブロック崩しが実行される。

また、カーネルのレイヤで実装しているが故の特徴として、ブロック崩しに失敗し、弾が画面下まで落ちてしまった場合、OSの動作が停止するようになっている。

kaOS



kaOSは「30日でできる！　OS自作入門」[1]の内容を参考にしながら、Cで記述された部分をC++で置き換えて作成しているOSである。

基本的にはC言語で手続き的に記述されたプログラムを構造化し、より機能拡張やメンテナンスがしやすい形で実装し直すことを目的としているが、それに加えてはりぼてOSの明確な問題点を解決する形で新たな実装を加えている部分もある。代表的なものはメモリ管理であり、後のメモリ管理の項でに記述する。

上図で示しているのは、現在制作した部分のオブジェクトの関係を示すUMLクラス図である。基本的には処理の起点となるオブジェクトを`KernelMain`とし、そのメンバとして、OSの細かい責務を行うオブジェクトが存在するというシンプルな構成となっている。そのため、以下のように`KernelMain`のコンストラクタ内でOSの初期化を簡潔に記述できるようになる。

```
// kaOS/bootkaos
KernelMain::KernelMain(const BOOTINFO& bootinfo) :
    segment_(Segment::SEGMENT_DESCRIPTOR_TABLE_ADDRES, 0x0007ffff, 0x00280000),
    interupt_(Interupt::INTERUPT_DESCRIPTOR_TABLE_ADDRES,
        Interupt::PIC0_FIRST_INTERUPT_NUMBER,
Interupt::PIC1_FIRST_INTERUPT_NUMBER),
    palette_(Palette::simplePalette()),
    screen_(bootinfo.vram, bootinfo.scrnx, bootinfo.scrny, 99),
    kbc_(),
    keyboard_(&kbc_, interupt_, segment_.getKernelSegmentNumber()),
    mouse_(&kbc_, interupt_, segment_.getKernelSegmentNumber())
{
    palette_.set();
    screen_.drawBack();
    io_sti();
}
```

メモリ管理を行うオブジェクトMemoryは、カーネル内でC++のnew演算子やdelete演算子の機能をグローバルに使用できるようにするためグローバルなインスタンスを生成して責務をこなすようになっている。

基本的には継承は使用せずにオブジェクトの関係は簡潔なものとなっているが、一部で異なったOSの機能間での依存性を生まないようにインターフェースを用いている部分もある。

プロセス管理

一般的なプロセス管理

現在一般的に使われているOSは、ほとんどが複数のプロセスを同時に動かすことが出来る。この複数のプロセスを同時に実行するための仕組みはOSが実現する必要があり、この責務をプロセス管理と呼ぶ。

このとき、重要なのはユーザー空間のプログラムからは一つのプロセスが継続的に動いているようにレジスタやメモリ空間を扱える一方で、OSを使うユーザーからは複数のプロセスが同時に動いているように仮想化を行うことである。

ユーザー空間のプログラムからレジスタやメモリ空間を自由に使うためには、単純にそれぞれのプログラムごとに仮想的なレジスタと、それぞれの固有のメモリ空間とを与えればよいことになる。このとき、仮想的なレジスタはメモリに置くことで、他のプロセスが上書きすることを防ぐことができる。プロセスは、実行するプログラムだけでなく、プログラムごとのメモリ空間や仮想レジスタを含めたもののことであるといえる。

複数のプロセスを同時に実行するマルチプロセスは、厳密な意味で同時にプロセスを実行するわけではなく、実際には実行するプロセスを短時間で切り替えることによって同時に実行しているかのように見せることが出来る。これによって、CPUの数に依存せず多くのプロセスを同時に動かすことが出来る。

この実行プロセスの切り替えはコンテキストスイッチと呼ばれ、コンテキストスイッチによってどのプロセスに切り替えるのかを決定することをスケジューリングと呼び、様々なアルゴリズムが存在する。

スケジューリングのアルゴリズムにはいくつかの種類があり、それらが組み合わされることもある。

プロセスの実行順は単純に先に行う必要が出てきたものから実行するFIFOや、プロセスの終了までの時間が最も短いものから実行するSJFで決定される場合や、何らかの基準によってつけられた優先度に従って実行される場合もある。

また、それぞれのプロセスが実行される時間は、一定に区切られる場合もあれば、プロセスが終了するまでコンテキストスイッチをしない場合や、割り込みによって実行可能となったプロセスが現れた時にコンテキストスイッチする場合もある。

プロセスには、プログラムを実行するための様々なデータが必要であり、それらは専用のデータ構造によって管理される。これには、一般にプログラムの実行可能な命令を格納するコード領域や、データ領域、スタック領域、管理用の情報などが含まれている。このデータ構造はIDによって一意に識別される場合もある。

hariballOS（hariboteOS）のプロセス管理

メモリ空間について

プロセス管理を実装する際には、上述したプロセスごとのメモリ空間が必要となる。

はりぼてOSにおける、ユーザー空間のプログラムを実行するプロセスごとのメモリ空間は一つではなく、コード領域、データ領域、スタック領域にあたる3つのメモリ空間が確保される。この中でコード領域とデータ領域に関しては、「30日でできる！　OS自作入門」[1]の中でコードセグメント、データセグメントと呼ばれ、セグメンテーションと呼ばれるメモリ領域の分割手法で用いて分割している。セグメンテーションについての詳細はメモリ管理の項で解説するが、プロセスにおいてセグメントはそれぞれに読み書きの権限を与えることが出来、セキュリティの面で役に立っている。

ユーザー空間のプログラムのコード領域とデータ領域の確保は以下のプログラムで行っている。

```
// hariball/haribote/console.c
q = (char *) memman_alloc_4k(memman, segsiz);
task->ds_base = (int) q;
set_segmdesc(task->ldt + 0, finfo->size - 1, (int) p, AR_CODE32_ER + 0x60);
set_segmdesc(task->ldt + 1, segsiz - 1, (int) q, AR_DATA32_RW + 0x60);
```

`set_segmdesc()`という関数を用いてメモリ領域をセグメントとしてCPUに伝え、それによって`p`から実行ファイルのサイズ分をコードセグメント、`q`から実行ファイルに記述されたサイズ分をデータセグメントとしている。また。データセグメントの先頭アドレスはタスクの情報を保持するtask構造体のメンバであるds_baseに代入している。

これらのアプリごとのセグメントはLDTと呼ばれる仕組みを用いており、読み書き権限はOSのものとは異なるが、アプリ内から書き換えることはできないようになっている。

プロセスごとのスタック領域は、ユーザー空間のプログラム用のものはデータセグメント内に存在することになるが、カーネル空間で動作するプロセス上のプログラムはプロセス生成時に設定したスタック領域を用いて動作する。後者は、この後に記述するTSSの設定時に確保される。

レジスタについて

仮想的なメモリ上のレジスタも、メモリ空間同様プロセスごとに用意しなくてはならない。

これは、TSSと呼ばれるセグメントに格納しておくことで、x86のfarモードのジャンプ命令を用いてタスクスイッチをすることが可能となる。

TSSは以下のような構造体を用いて中身のデータを記述することが出来る。

```
// hariball/haribote/bootpack.h
struct TSS32
{
    int backlink, esp0, ss0, esp1, ss1, esp2, ss2, cr3;
    int eip, eflags, eax, ecx, edx, ebx, esp, ebp, esi, edi;
    int es, cs, ss, ds, fs, gs;
    int ldtr, iomap;
};
```

これらは、ほとんどがレジスタ内容を保存するためのものであるが、1行目と4行目のメンバは、特にプロセスの設定を行うためのものとなっている。

TSSを設定しておくことによりアセンブリ言語上の記述は以下のように単純にすることが出来る。

```
; hariball/haribote/naskfunc.nas
_farjmp:        ; void farjmp(int eip, int cs)
        JMP     FAR [ESP+4]
        RET
```

このコードははりぼてOSで用いられているアセンブラであるNASK向けに書かれたものである。

仮引数である`eip`の値はアドレス`ESP+4`に格納されているため、`[ESP+4]`で読み出すことが出来る。また、もう一つの仮引数である`cs`は`[ESP+8]`で読みだすことが出来るが、`JMP FAR`命令に`[ESP+4]`を指定することで、さらに2バイト分読みだしてコードセグメントの値として用いることが出来る。これによって`cs`で指定したTSSのプロセスに処理が移行する。

スケジューリングについて

はりぼてOSは、割り込みや処理の終了ではなく、基本的には時間の経過によってもコンテキストスイッチを行い、プロセスにはコンテキストスイッチ時に参照される優先順位が付与されている。

まず、コンテキストスイッチの基準となるプロセスごとの実行時間と優先順位はプロセスの登録時に設定する。その時のプログラムは以下である。

```
// hariball/haribote/mtask.c
void task_run(struct TASK *task, int level, int priority)
{
    if (level < 0)
    {
        level = task->level;
    }
    if (priority > 0)
```

```
    {
        task->priority = priority;
    }

    if (task->flags == 2 && task->level != level)
    {
        task_remove(task);
    }
    if (task->flags != 2)
    {
        task->level = level;
        task_add(task);
    }
    taskctl->lv_change = 1;
    return;
}
```

第一引数で設定したいプロセスの参照をポインタとして渡し、そこに設定を行う。

第二引数としてはプロセスの実行優先度を与える。実行優先度は`level`と呼ばれ、いくつかの段階に分けられる。段階ごとにプロセスが複数あり、もっとも優先度の高い`level`に存在するプロセスで時間を共有してプロセスを実行していく。このとき、優先度の`level`が最も高いプロセス以外のプロセスは実行されず無視される。

第三引数はプロセスを実行し、コンテキストスイッチが起こるまでの時間を表している。この値が大きいほど、一度に長い時間プロセスが実行されることになる。これも一つのプロセス実行の優先度の一つの表現と考えることもできるため、仮引数名は`priority`となっている。

これらの優先度と実行時間を参照して、レジスタの項で説明した`farjmp()`関数を用いてコンテキストスイッチを行うことでスケジューリングを行うことが出来る。

プロセスごとの実行時間の終了はタイマ割り込みによって実装されている。はりぼてOSでは割り込みハンドラからコンテキストスイッチの関数を直接呼び出してコンテキストスイッチを行っている。

また、プロセスの停止（スリープ）はコンテキストスイッチ時に参照される`level`ごとのプロセスの集合から停止させたいプロセスを取り除くことで実現できる。停止からの再実行は、はりぼてOSでは該当プロセスに対する割り込みの発生時ということになっている。

メモリ管理

一般的なメモリ管理

コンピュータの処理の実行においてメモリの役割は大きく、CPUがメモリに対して読み書きを行うことでコンピュータの責務の多くを実現している。しかし、多くのプロセスを同時に実行する必要性などから、限られたメモリというデータの領域をどのように配分してプログラムで使用するかを決定する必要がある。この責務は、基本的にOSが負うこととなっている。

メモリは、プロセスごとにそれぞれ固有のメモリ空間が0番地から存在するかのように仮想化されていることが一般的であり、これによってプログラムは物理的なメモリのレイアウトや空き容量を知る必要はなくな

る。

もしも物理的なメモリ空間をプロセスごとに連続した領域で分割していく場合、フラグメンテーションなどの問題が発生し、メモリ領域がさらに圧迫されることもある。

そこで、一般的にメモリ管理にはページングという手法が用いられる。ページングは物理的なメモリ空間を固定長に分割しそれらと、仮想的なメモリアドレスで表されるメモリ空間とを結び付け、その結びつきをテーブルとしてMMUと呼ばれる領域で管理する手法である。このテーブルはページテーブルと呼ばれる。

単純なページングにはデメリットもいくつかあり、そのひとつがページテーブルの大きさである。ページテーブルは仮想的に作られたメモリ空間に存在するすべての固定長の領域であるページに対してエントリを持つ必要があり、その長さはページの大きさにも依存するが、少なくとも仮想メモリ空間の定数倍の大きさになる。ページの大きさを大きくすることもできるが、そうなると無駄に使用されるメモリ領域が増えることとなる。

そこで、ページングに代わるいくつかの手法が提案されている。上述のページングを階層化して行う階層化ページテーブルや、可変長のメモリ空間を用いてメモリ管理を行うセグメンテーションなどがその例である。

hariballOS（hariboteOS）のメモリ管理

セグメンテーションについて

はりぼてOSでは、メモリ管理の手法としてセグメンテーションを採用している。「30日でできる！　OS自作入門」[1]の筆者は、自身のWikiのページ[2]にてセグメンテーションの利点を以下のように述べている。

> セグメンテーションがあれば、0番地から始まる領域を同時に複数使わせることができます。たとえばプログラムコード用、データ用、スタック用、共有メモリ用、共有ライブラリ用、などです。特に共有ライブラリではとても便利で、なぜならどの番地にロードされるか悩まなくていいからです。0番地だとあらかじめ決め打ちできます。

このように多くの場面でメモリマップを意識せずにセグメントとして分けられたメモリ領域を使うことのできる点をメリットの一つとして挙げています。

他にも筆者が過去に制作したOSであるOSASKを例にとって以下のようにも述べています。

> OSASKでは基本的にセグメンテーションによってメモリを切り分けます（厳密には分け合っているのはメモリ空間であってメモリではないですが）。だからタスクがいくつあってもメモリ空間は1つしか使いません。つまりCR3は一つしかないのです。これにより、どんなに頻繁にタスクスイッチしてもCR3が変わらないので、TLBはクリアされません。（中略）これによりメモリ効率はとてもよく、大変快適でした。 そして競合するどの自作OSよりも圧倒的に高速で、「セグメンテーションを使うと遅い」という思い込みを完全に打ち砕いたのでした。

このように、ページングではタスクスイッチ時にリセットされてしまうTLBに注目し、過去のOS制作の経験から、セグメンテーションが速度的な側面でもページングに勝る部分があることを示しています。

上記の理由も含め、今回のように初学者に向けた書籍においては、メモリ空間を0番地から始まると考えられる単純さや、全体の実装のコンパクトさから、セグメンテーションが適していると判断したと考えられる。

セグメンテーションを実装するためには、セグメントごとの情報をメモリに書き込み、そのアドレスと大きさをGDTRと呼ばれるレジスタに書き込む必要がある。

メモリに書き込む情報は以下のような構造体で表現できる。

```
// hariball/haribote/bootpack.h
struct SEGMENT_DESCRIPTOR {
        short limit_low, base_low;
        char base_mid, access_right;
        char limit_high, base_high;
};
```

多くが、メモリ領域の大きさについての値を格納するが、`access_right`では、セグメントごとのアクセス権限を設定することも可能であり、これによってカーネルとユーザー空間のメモリ領域を明確に定義することが可能となり、CPUによってそれが守られるようになる。

次に、セグメントに関する情報をGDTRに書き込む処理は、NASKのアセンブリ言語で以下のように記述される。

```
; hariball/haribote/naskfunc.nas
_load_gdtr:     ; void load_gdtr(int limit, int addr);
        MOV     AX,[ESP+4]
        MOV     [ESP+6],AX
        LGDT    [ESP+6]
        RET
```

`LGDT`命令によって指定したメモリから6バイトの値が読み込まれることを利用して、引数`limit`の下位2バイトと`addr`の値を6バイトに収めて渡している。

カーネル内のメモリ管理について

はりぼてOSでは、OSに依存するlibcの機能は使わず独自に実装する必要がある状態でC言語での開発を行うため、動的にメモリを確保する必要がある場合、メモリの一部を分割して管理する方法を自ら用意する必要があった。そのため、通常C言語では`malloc()`を用いて行うような処理を記述している部分がある。

この領域は、`MEMMAN`という構造体と、`MEMMAN`型の値を引数に取る一連の関数として実装されており、APIとしてユーザー空間にも同じ仕組みが提供されている。ユーザー空間で用いる場合は、管理する領域はプロセスのデータ領域となる。

```
// hariball/haribote/bootpack.h
struct FREEINFO
{
        unsigned int addr,size;
};
struct MEMMAN
{
        int frees, maxfrees, lostsize, losts;
```

```
        struct FREEINFO free[MEMMAN_FREES];
};
```

MEMMAN構造体は上記のようなメンバをもつ。管理するメモリの使用可能な領域の先頭アドレスと使用可能な領域の大きさを固定長の配列の中に格納している。この配列から、確保したい領域以上の長さを持つ使用可能な領域を探し、メモリの確保を行う。

このとき、メモリの確保と解放の繰り返しによってメモリはフラグメンテーションしていくが、使用可能な領域を持つ配列が固定長であるため、いずれ追跡できないメモリ領域が現れる可能性がある。その場合、追跡できなくなったメモリの領域の大きさと数をlostsizeとlostsに格納している。

kaOSのメモリ管理

セグメンテーションについて

kaOSでは、はりぼてOSにならい、メモリ管理にはセグメンテーションを用いている。

セグメントの設定は、はりぼてOSではグローバルにどこからでも行うことが出来たが、kaOSではSegmentというオブジェクトにその責務を与えている。

```
// kaOS/memory/segment.cpp
Segment::Segment(unsigned int segmentDescriptorTableAddress) :
    gdt_(reinterpret_cast<SegmentDescriptorTable *>
(segmentDescriptorTableAddress)),
    gdtSize_(1),
    kernelSegmentNumber_(0)
{
    new(gdt_) SegmentDescriptorTable();
    gdt_->set(gdtSize_++, 0xffffffff, 0x00000000,
SegmentDescriptorTable::SYSTEM_RW_AR);
}
unsigned int Segment::setKernelSegment(unsigned int limit, int base)
{
    gdt_->set(gdtSize_++, limit, base, SegmentDescriptorTable::SYSTEM_RX_AR);
    return gdtSize_ - 1;
}
```

Segmentクラスのコンストラクタによって、引数で指定されたアドレスを用いて配置newによってSegmentの設定を記述するメモリ空間を初期化し、GDTRへの読み込みを行っている。セグメントの設定は、上記のsetKernelSegment()に代表されるメソッドによって行うことが出来る。

カーネル内のメモリ管理について

kaOSはMinGWに付属するg++を用いて開発を行っているが、libc++を使うことが出来ないため、はりぼてOS同様動的メモリ確保を行うための仕組みを実装している。

この実装は、メモリの使用可能な領域を連結リストにして扱うことで、はりぼてOSでの、追跡不可能なメモリ領域が生まれることを防いでいる。

```
// kaOS/memory/memory.h
class  Memory
{
private:
    class FreeListElement
    {
    private:
        unsigned int size_;
        FreeListElement *prev_;
        FreeListElement *next_;
    }
    const unsigned int beginAddress_;
    const unsigned int endAddress_;
    unsigned int freeSize_;
    FreeListElement freeListEntry_;
}
```

上記のプログラムは、メモリ管理を行うクラスMemoryのヘッダーでの宣言の一部である。

Memoryは内部実装としてFreeListElementというクラスを実装しており、このクラスがメモリ領域の確保や解放をするためのメモリ領域の単位を表している。

Memoryは連結リストとなっているFreeListElementの先頭のインスタンスを所有しており、このインスタンスから常に連結リストをたどることが出来る。また、コンストラクタで与えられるメモリ領域の開始アドレスと終了アドレスを保持している。

FreeListElementは連結リストの一つの要素でありながら、そのインスタンスは実際に確保・開放されるメモリ領域の先頭部分に位置する。この性質によってメモリの使用可能な領域が発生する度に連結リストの要素が作成されるため、追跡できないメモリ領域が発生することはない。

```
void *Memory::allocate(unsigned int size)
{
    for (FreeListElement *elem = freeListEntry_.getNext();
        reinterpret_cast<unsigned int>(elem) < endAddress_ && elem != nullptr;
        elem = elem->getNext())
    {
        if (elem->allocate(size, endAddress_))
        {
            freeSize_ -= elem->getTotalUseMemorySize();
            return reinterpret_cast<void*>(reinterpret_cast<unsigned int>(elem) +
sizeof(unsigned int));
        }
    }
    return nullptr;
}
```

メモリ領域の確保は上記のような実装がされている。メンバとしてもつ連結リストの先頭要素からイテレーションを開始し、最終アドレスおよび末尾まで走査して確保できる領域を探している。メモリ領域が確保で

きる場合、その領域の大きさの情報を残すために、領域の大きさを格納する部分だけをずらして使用できるメモリ領域の先端を返り値として返している。

ファイル・デバイス管理

一般的なファイル・デバイス管理

hariballOS（hariboteOS）のファイル・デバイス管理

ファイル管理について

デバイスの管理について

割り込みについて

kaOSのデバイス管理

デバイスの管理について

割り込みについて

参考文献

1. 「30日でできる！　OS自作入門」（川合秀実）マイナビ出版 2018年11月19日発行
2. 「KH-FDPL（けいえいち・えふでぃぴーえる）のWiki - セグメンテーションの夢はどうなったのか？」http://khfdpl.osask.jp/wiki/?advcal20161206 7/25参照