ゲーム制御のためのメモリ管理方針

– ゲーム制御のためのメモリ管理の考え方 –

2014 年 2 月 24 日 初稿

板垣 衛

■ 改訂履歴

稿	改訂日	改訂者	改訂内容
初稿	2014 年 2 月 24 日	板垣 衛	（初稿）

■ 目次

■ 概略

ゲームシステムのメモリ割り当てについて考察し、最適なメモリ管理の方針を固め、メモリマネージャ設計の土台とする。

本書で固めた方針に基づいて、別紙の「ゲーム開発のためのメモリアロケータ」を設計する。

■ 目的

本書は、メモリマネージャの設計にあたり、考慮すべき事項を明確にし、設計方針を固めることを目的とする。

システム要件を確定するような資料ではなく、その前の検討事項のまとめである。

■ メモリ管理の課題と方針

メモリマネージャを設計するために、考慮すべきメモリ管理の課題とその対応方針を検討する。

▼ メモリ制限の方針：制作者がコストの上限を意識するために

マップ、メインキャラ、敵キャラ、ボス、NPC、エフェクト、イベント、サウンド、メニューなど、それぞれメモリには制限が必要であり、コンテンツ制作者はそれを意識して制作できなければならない。

● 【不採用】方法①：メモリマネージャのインスタンスを多数用意

メモリの区分ごとにメモリマネージャのインスタンスを設け、それぞれのサイズのサイズをあらかじめ決めておく。

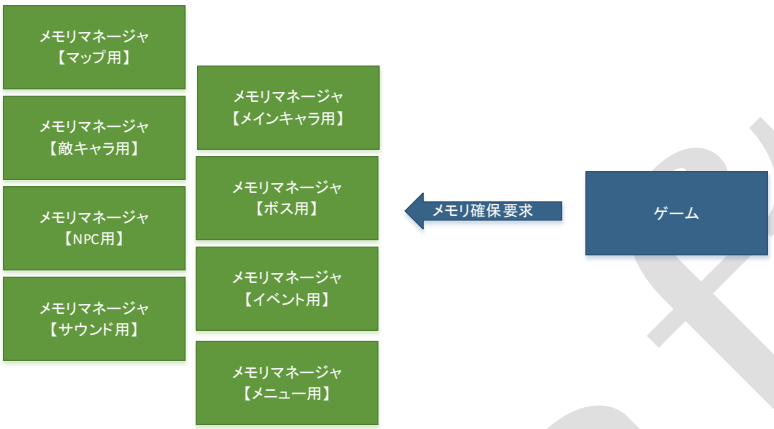
- 実装方法： メモリ確保の際、メモリ区分もしくはメモリマネージャのインスタンスを指定する。
- 長所： ・ 確実な制限が可能。
  ・ メモリ確保数が分散し、メモリマネージャの処理効率が良い。

・ フラグメンテーションの影響を分散できる。

➢ 短所： ・ 状況に応じて制限を変える事が難しい。

・ 区分が多くなるとそれぞれの空き領域の合計が大きくなり、メモリの使い方として無駄が多い。

イメージ：



メモリ区画ごとに多数のメモリマネージャのインスタンスを用意

● 【不採用】方法②：メモリ使用状況をゲーム側で集計して画面表示

大きなメモリ空間を共有し、「カテゴリ」ごとにメモリ使用量を集計する。

デバッグ機能により、画面に区分ごとのメモリ使用状況を表示し、制限を超えたときに警告を出すなどする。

➢ 実装方法： メモリ確保の際、ゲーム側で用意した集計機能付きのアロケータを使用し、カテゴリを指定する。

➢ 長所： ・ メモリの無駄が少ない。

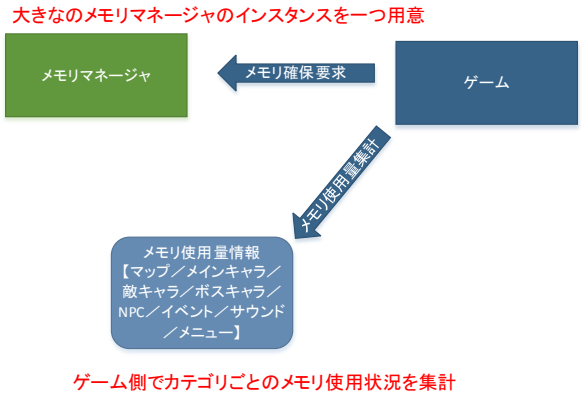
➢ 短所： ・ 不確実。

・ 一つのメモリマネージャで多数のメモリ確保を扱う為、処理効率が劣化する。

・ 実装が手間。

・ フラグメンテーションの影響が全体に及ぶ。

イメージ：

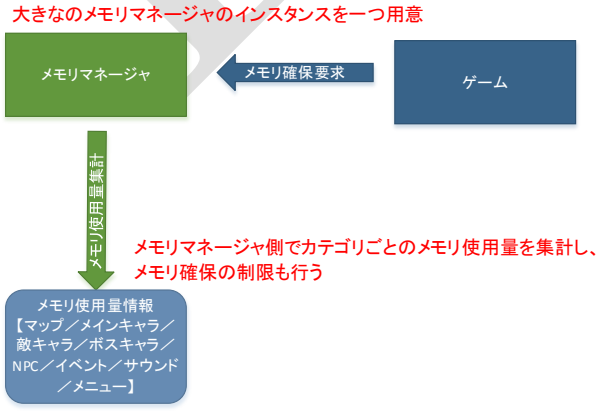


● 【不採用】方法③：メモリマネージャに「メモリ制限機能」を加える

メモリマネージャに対して、「カテゴリを指定してメモリを確保する機能」と、「カテゴリのメモリ制限を設定する機能」を実装し、制限を超えるメモリ確保をできなくする。

- 実装方法： メモリマネージャにカテゴリごとの集計と確保制限の機能を実装し、メモリ確保の際、メモリ区分を指定する。
- 長所：
  - 確実な制限が可能。
  - メモリの無駄が少ない。
- 短所：
  - 一つのメモリマネージャで多数のメモリ確保を扱う為、処理効率が劣化する。
  - フラグメンテーションの影響が全体に及ぶ。
  - 実装がやや手間だが、一度作ればよいので大きな問題ではない。

イメージ：



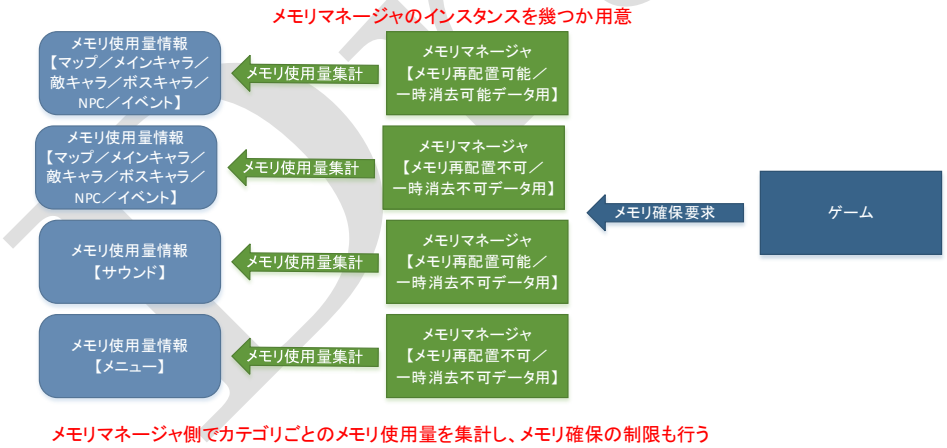
● 【不採用】方法④：「方法①」と「方法③」の組み合わせ

メモリマネージャのインスタンスを大きなまとまりに区分けし、その中で「カテゴリ」ごとの集計と制限を行う。

インスタンスの区分けは、「メモリ再配置可能なメモリ」、「（ムービー再生時などで）大きな連続領域の要求時に全消去可能なメモリ」といったことを基準にする。

- 実装方法： メモリマネージャにカテゴリごとの集計と確保制限の機能を実装た上で、複数のメモリマネージャのインスタンスを扱う。一つ一つのメモリマネージャは、それぞれ複数のカテゴリに対応する。メモリ確保の際は、メモリ区分が指定され、メモリ区分に応じて適切なメモリマネージャ（のインスタンス）からメモリ確保を行う。
- 長所：
  - 確実な制限が可能。
  - メモリの無駄が少ない。
  - フラグメンテーションの影響を分散できる。
  - メモリ確保数がやや分散し、メモリマネージャの処理効率が少しだけよくなる。
- 短所：
  - 実装がやや手間。

イメージ：



● 【採用】方法⑤：「方法④」の改良版

基本的に「方法④」を踏襲するが、必要なインスタンスの数を減らす（完全に一つにはできない）。

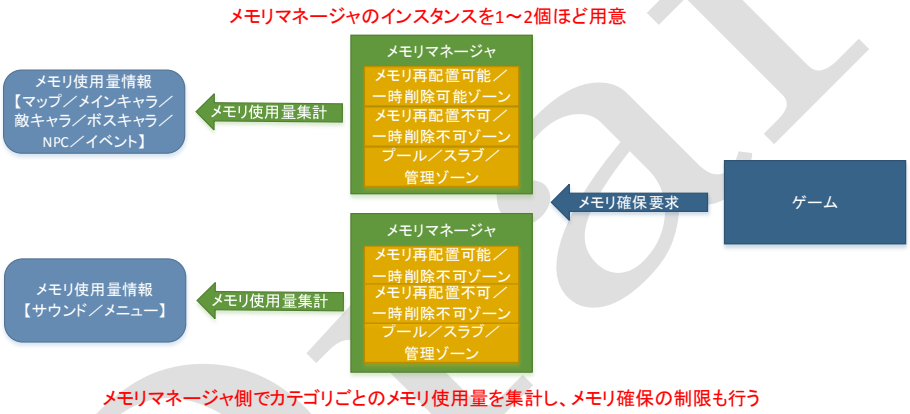
メモリマネージャ内に物理的な区画（ゾーン）を設けることで対応する。

- 実装方法： メモリマネージャに物理的な区画（ゾーン）を設ける。「カテゴリ」のグ

ループのような位置づけで扱い、メモリ確保時はカテゴリだけの指定で良いようにする。

- 長所：
  - 確実な制限が可能。
  - メモリの無駄が少ない。後述する「プールアロケータ」や「スラブアロケータ」を、複数の区画が共通利用できるので、「方法④」よりも更に効率的になる。
  - フラグメンテーションの影響を分散できる。
- 短所：
  - 実装がやや手間。
  - 一つのメモリマネージャで多数のメモリ確保を扱う為、処理効率が劣化する。ただし、メモリノードの高速検索アルゴリズムを採用することでこの問題を解消する。

イメージ：



▼ ムービー対応の方針：大きな連続領域の確保のために

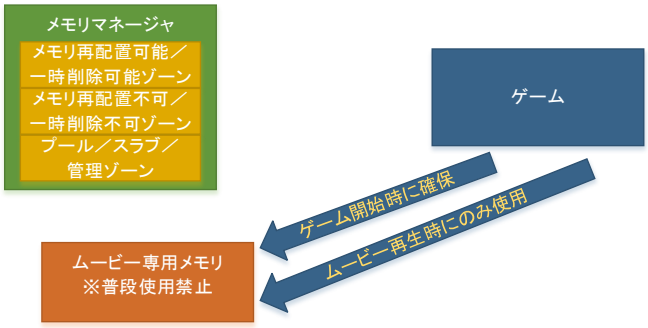
ムービー再生時は大きな連続領域を必要とするため、あらかじめムービー再生に備えた何らかの対処をしておかないとメモリを確保できない。

● 【不採用】方法①：常にムービー用のメモリを確保しておく

ムービー再生用のメモリは常に確保しておく。

- 実装方法：　ゲーム開始時にメモリを確保する。
- 長所：
  - 確実。
  - 実装が簡単。
  - ムービー再生中に、他のリソースに影響を与えない。
- 短所：
  - ムービー時以外のメモリがもったいない。（大きな問題）

イメージ：

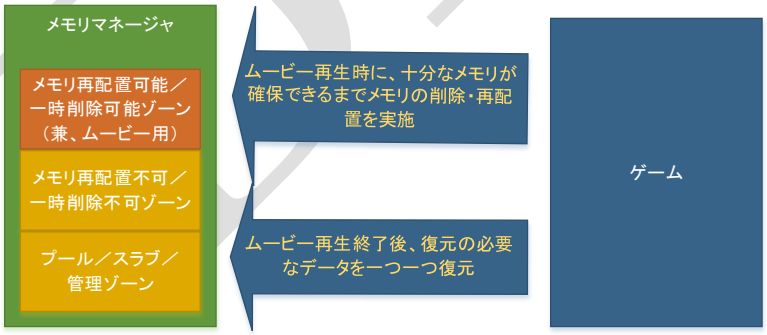


● 【不採用】方法②：ムービー再生時にメモリを空ける

ムービー再生時に、データ削除やコンパクションを利用し、何とかして大きな連続領域を空ける。

- 実装方法： ムービー再生時に、ムービーのためのメモリ確保が成功するまでデータの削除やコンパクションを行う。
- 長所： ・ムービー時以外にもメモリを無駄にしない。
- 短所： ・不確実。（大きな問題）
  - ・実装が手間。
  - ・処理に手間がかかる。
  - ・復元（前の状態に戻す）処理に手間がかかる。

イメージ：



● 【採用】方法③：ムービー時に全クリア可能なメモリを用意

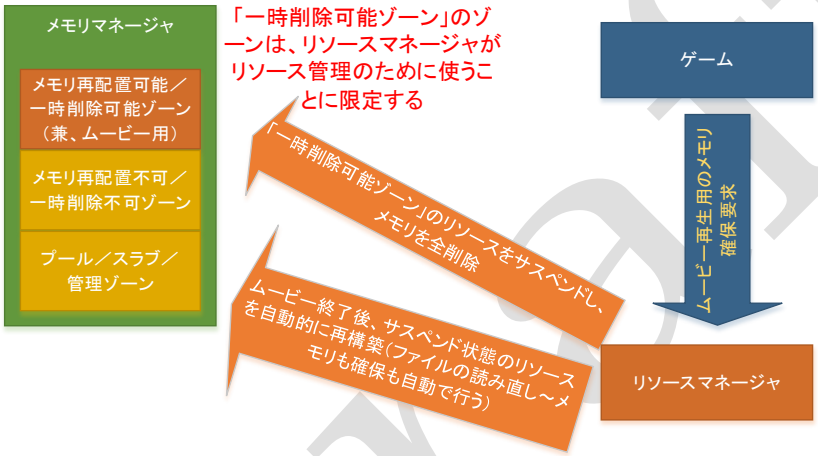
通常は他の用途で使用していながら、ムービー時には全クリアして良い、物理的なメモリ領域（もしくはメモリマネージャのインスタンス）を用意する。

- 実装方法： 「リソースマネージャ」の機能で「リソースの一時削除と自動再構築」が

できることを前提に、「一時削除が可能なグラフィックリソース専用のメモリ領域」を設ける。（別紙の「開発の効率化と安全性のためのリソース管理」でも説明）

- 長所：
  - （削除の手間はかかるが）確実。（大きな問題の解決）
  - ムービー時以外にもメモリを無駄にしない。（大きな問題の解決）
- 短所：
  - 実装が手間。（リソースマネージャとそれを使用したシーンマネージャなど、システムをしっかりと固める必要がある）
  - ムービー前にメモリを空ける処理と、ムービー後にリソースを復元する処理にやや手間（時間）がかかる。

イメージ：



▼ メモリ効率向上のための方針：少しでも無駄のないメモリ確保を行うために

Linux のメモリ管理システムなどを参考に、幾つかのメモリ管理方法に基づいて、ゲームに最適なメモリ管理方法を検討する。

● 【部分採用】検討事項：64bit 対応

メモリのアドレスやサイズを 64bit で扱うことで、4GB を超える大きなメモリ空間を扱うことができる。

しかし、単純な 64bit アドレスの利用は不採用とする。

現状、4GB のメモリ空間で十分なことが多いため、メモリマネージャ内でのアドレス情報（ポインタ）、サイズ情報を 4 バイトに抑え、少しでもメモリの消費を抑える。

4GB 以上のメモリを扱う場合、メモリマネージャのインスタンスを分けて扱う。

アドレスは、メモリマネージャが管理するメモリ領域の先頭からのオフセットで扱い、加算した値を実アドレスとする。メモリマネージャの外部とは実アドレスでやりとりする。

【不採用】検討：32bit で 16GB 拡張

メモリマネージャ内に物理的な区画（ゾーン）を設ける場合、区画ごとに先頭の実アドレスを管理すると、4GB を超える領域を効率的に扱る。

ただし、この案は採用しない。

メモリノードの検索効率を考え、全区画のノードを連結管理することを考えたいため、メモリマネージャ内では単純なアドレッシングとしたい。

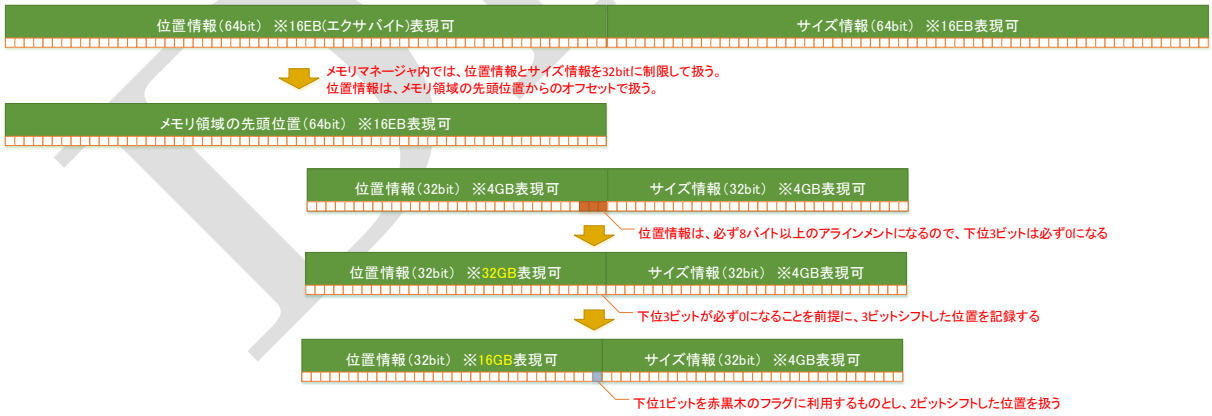
また、区画の境界を動的に変動させるような仕組みを導入したい場合も障害になる。

【採用】検討：32bit で 8GB 拡張

確保するメモリが実質 8 バイトでアラインメントされることを考えると、内部的なアドレス（とサイズ）は 3 ビットの右シフトが可能になり、32GB のアドレス空間を表現できるようになる。

ヒープメモリのチャンク連結に赤黒木（red-black tree）アルゴリズムを採用することを考慮すると、1 ビットはフラグに利用し、アドレスのシフトを 1 ビットに抑え、16GB のアドレス空間を扱うような仕組みも考えられる。

イメージ：



● 【部分採用】検討事項：仮想アドレス

Linux や Windows などのマルチプロセス OS は、CPU の仮想アドレス機能を利用し、プロセスのメモリ空間を保護する。

仮想アドレスの仕組みにより、物理メモリと異なるアドレス空間を扱うことが可能となる。これにより、マルチプロセス環境では、それぞれのプロセスが同じアドレスの

メモリにアクセスしても、実際の物理メモリは異なるため、競合することがない。

Linux のように、4GB 以上のメモリをサポートする 32bit OS は、この機能を使って、物理メモリを 32bit のアドレス空間に割り付けて扱う。

この仕組みを利用すると、物理メモリを大きく超えるメモリ空間を扱うこともできる。

ページ単位（4KB）で実際にメモリを使用する時にアドレスを割り当て、不要になったら返却する。これにより、ページ単位でのメモリの再アドレッシングが可能となり、断片化が発生しても、アドレスを再割り当てして大きな連続領域として確保し直すといったことも可能となる。

しかし、この仕組みの単純な利用は不採用とする。

仮想アドレスと物理メモリのマッピングを管理するための「メモリリージョン」の管理が必要となり、管理情報が増え、処理が複雑化する。

内部で複数のメモリ区画を扱うことを考えた場合、それぞれの区画に「どれだけ空きがあるのか？」を判定するのも難しくなる。

【可能なら採用】検討：拡張メモリ

全面的な仮想アドレスの採用とせず、「いざという時の拡張用」にだけ仮想アドレスを利用可能とする。

各メモリ区画内の空き領域が再アドレッシングされることはない。

全区画のどれにでも再割り当て可能なメモリとして予約しておき、範囲を超えそうなメモリ区画の領域を自動拡張する。

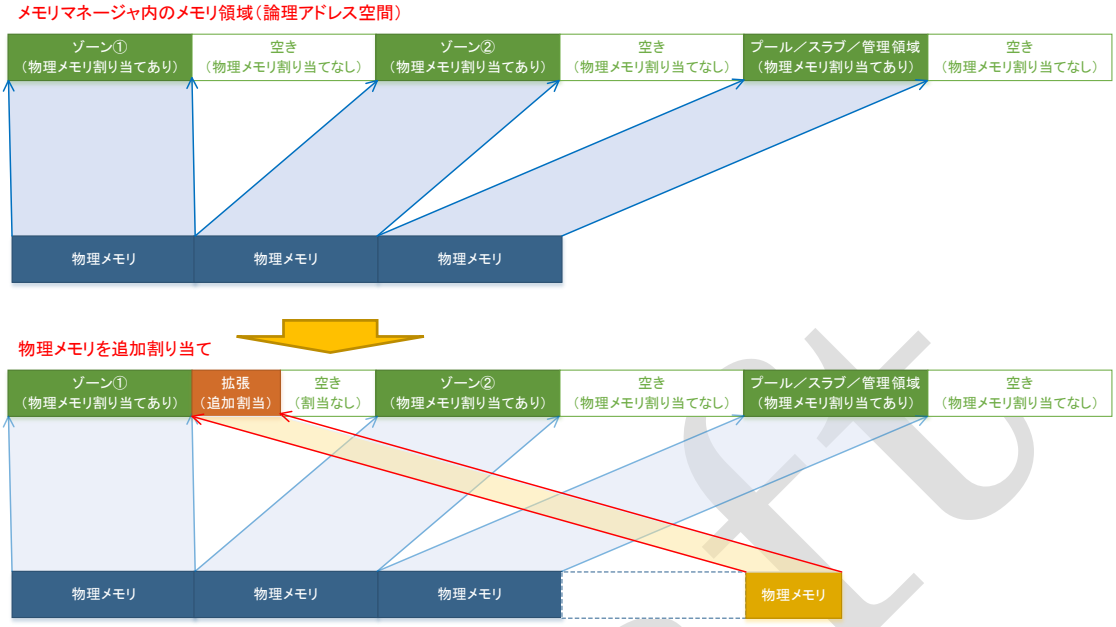
各区画の最初のメモリ割り当て時点で、十分な間隔を空けて仮想アドレスと初期物理メモリの割り当てを行っておき、必要に応じてリニアなアドレス空間にメモリを割り当てる。一度追加割り当てしたメモリは、その区画のメモリが全消去されない限り解放されない。

拡張可能な最大サイズや、一回に拡張するサイズもあらかじめ区画ごとに決めておく。

当然メモリリージョンの管理も必要になるが、わずかな管理となる。

このような形で、少しだけ「メモリの制限を緩くする」ことにより、メモリ不足の問題が発生した時にもハングせず、メモリ使用状況を把握することができるため、制作効率がよくなる。

イメージ：



【補足】Linux の「メモリリージョン」の仕組みは、プロセスメモリ空間の管理に用いられるものである。そのため、本来メモリ管理の順序としては、以降で検討事項にあげる「ゾーン」や「ページ管理」の後にくるものである。
しかし、本書は「ゲーム」という一つのプロセス内で用いるメモリマネージャを検討するものであるため、「仮想アドレスを意図的に扱うか？」という決定は、最初に検討すべきこととなる。

【参考】仮想メモリの最少単位（ページサイズ）は CPU の機能によって決まるが、使用する CPU アーキテクチャを前提に、各 OS でサイズを規定している。Linux や Windows では 4KB である。参考までに、PS3 には 1MB 単位の仮想メモリを割り当てる機能があった。

- **【採用】検討事項：ゾーン（ゾーン・アロケータ）**

Linux のメモリ管理では、x86(32bit)で三つ、x64(64bit)で二つの「ゾーン」を管理しする。メモリ使用の目的に合わせて物理的なメモリの「ゾーン」が大きく区切られている。

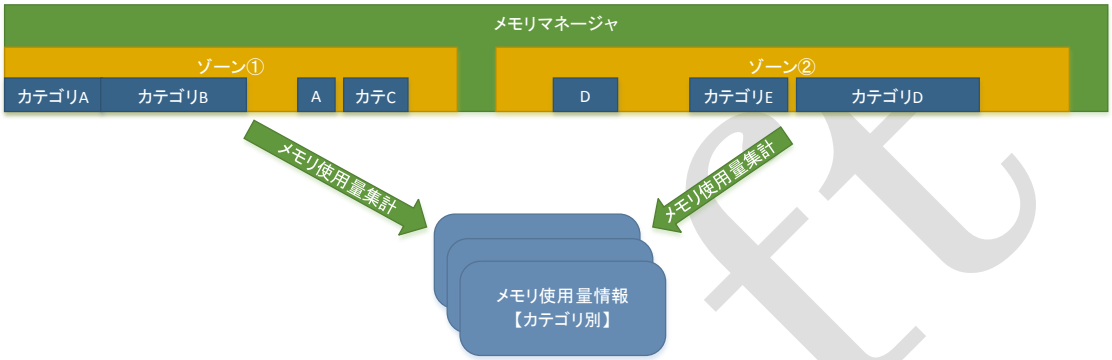
この考え方を真似て、「ゾーン」の仕組みの導入を採用する。

前述の「メモリ制限の方針」にも示したとおり、一つのメモリマネージャのインスタンス内で、メモリの物理的な区画を設ける。その区画を「ゾーン」と呼ぶ。

「カテゴリ」は、「メモリの論理的な制限」として扱う。

ゾーンとカテゴリは階層関係にあり、一つのゾーンには複数のカテゴリがある。カテゴリには物理的な区画がないため、一つのゾーンの中に多数のカテゴリのメモリが入り乱れるが、それぞれの合計サイズを管理し、制限を超えると、物理的な空きがあっても確保に失敗する。

イメージ：



【参考】x86 と x64 でゾーンの数が違うのは、x86 の三つめのゾーンが「High Memory」というゾーンで、32bit より大きな物理メモリを扱うためのものであるため。
そのゾーンでは、大きな物理メモリを 32bit アドレス空間にマッピングして扱うが、x64 ではそもそも 64bit アドレス空間を扱うので、必要が無い。
なお、残りの二つは、「DMA」ゾーン（0~16MB の空間）と「Normal Memory」ゾーン（x86 で 16~896MB の空間）。「DMA」ゾーンは、古い ISA デバイスが 16MB までしかアクセスできないので分けられている。

● 【部分採用】検討事項：ページ管理

Linux のメモリ管理では、x86 系の CPU のメモリ管理に合わせ、4KB のページサイズでメモリを分割して扱う。ゾーン・アロケータがゾーン内のメモリをページ単位で割り当てる。

しかし、単純なページ管理はゲーム用のメモリマネージャを無駄に圧迫するだけなので採用しない。

各「ゾーン」のメモリは、完全にヒープ領域としてのみ扱い、ページの管理は行わない。

【採用】検討：「ゾーン」以外のメモリのページ管理

「ゾーン」以外に、「プールアロケータ」、「スラブアロケータ」、「個別メモリ管理情報」（ゾーンに確保した個々のメモリノードの管理情報など）といったメ

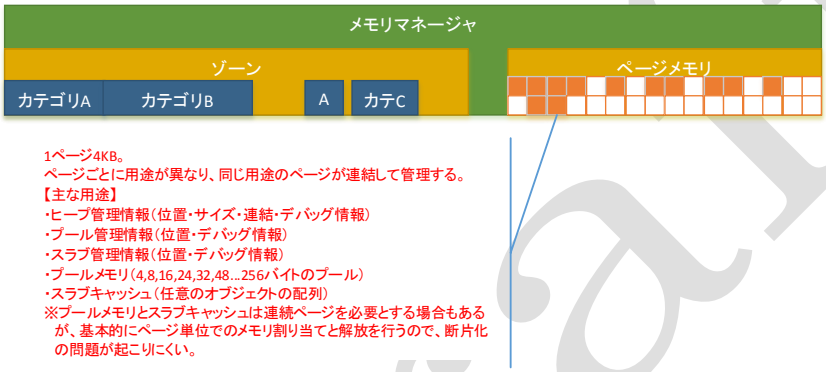
モリの管理を併用することを考慮し、かつ、これらのメモリ配分をランタイムで流動的に扱うために、ページメモリを管理する。

自由なサイズのメモリ確保が可能なヒープメモリと異なり、ページごとに用途を限定し、それぞれの用途に合わせた固定メモリの配列を扱う。

このメモリ空間は最大でも 1GB ほどの空間を上限に扱うものとし、下記の「バディシステム」で管理する。

また、ゾーンは 1MB アラインメントなどの大きなアラインメントが必要になることも考慮し、このページメモリ区画は、メモリ管理領域全体の最後方に配置して扱う。

イメージ：



【可能なら採用】検討：「ゾーン」のメモリ境界の変動

仮想アドレスを利用したゾーンのメモリ拡張を「可能なら採用」とするが、それができない場合、ゾーンのメモリ境界を変動する仕組みを検討する。

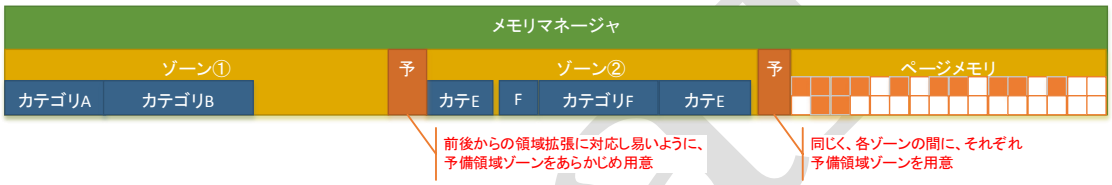
各ゾーンはリニアなメモリ空間に配置し、それぞれアドレスの先頭から順に使用する。ゾーンのメモリが足りなくなった場合に、一つ前に配置されているゾーンの最後方の空きを削り、メモリ境界を動かして区画のサイズを変動する。また、一つ後ろのメモリも前方に空きがあれば同様に境界を動かすことが可能。ページ管理領域も同様の変動が可能。なお、変動可能なメモリサイズにはあらかじめ制限を設け、変動するサイズはページサイズ（4KB）の倍数とする。

これにより、「メモリ不足の警告を出しつつもなるべくゲームを止めない」ということを実現する。

各ゾーンの間にあらかじめ「予備領域ゾーン」を挟むのも可。これはゲーム側でゾーンの設定を工夫することで対応できる。

ゾーンの領域拡張のイメージ：

あらかじめ領域拡張に備えたゾーン配置のイメージ：



【参考】仮に 4GB のメモリ空間を 4KB に区切ったページで管理すると、4GB÷4KB=約 100 万（1M）ページの管理が必要となる。仮に一ページにつき 16 バイトの管理情報を必要とした場合、それだけで 16MB のメモリが必要になる。1GB のメモリ空間なら 4MB。

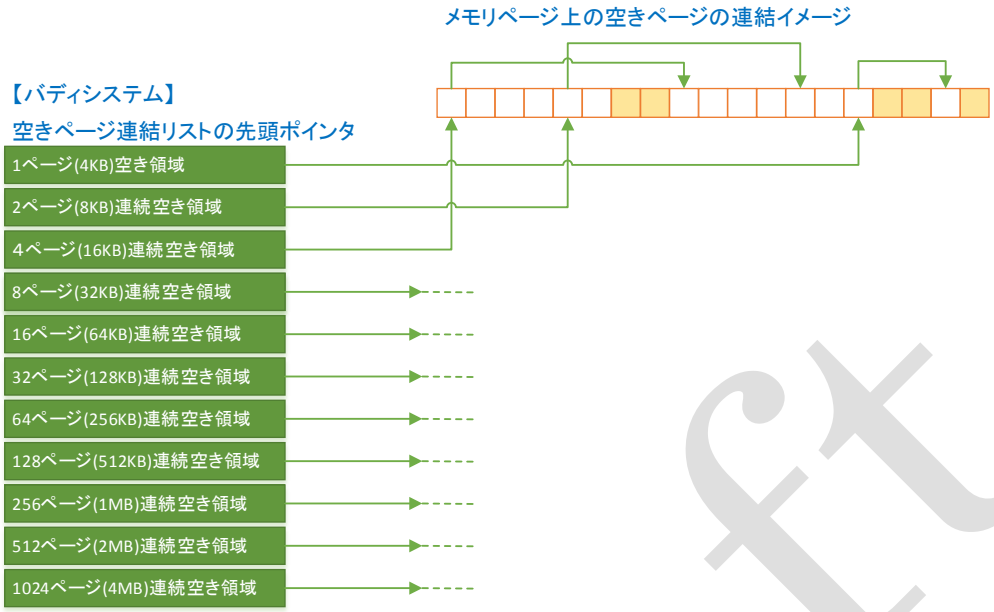
【参考】Linux のメモリ管理システムは、「ページ」と「ページフレーム」を扱う。どちらも 4KB のメモリ空間のことだが、後者は物理的なメモリを指し、ページの論理アドレスに割り当てて（マッピングして）扱う。本システムでは、ページフレームを意識することは特になく、すべて「ページ」と表記する。

- **【採用】検討事項：バディシステム**

Linux のメモリ管理では、「空きページ」の管理方法として、ページの断片化に強い「バディシステム」を採用している。

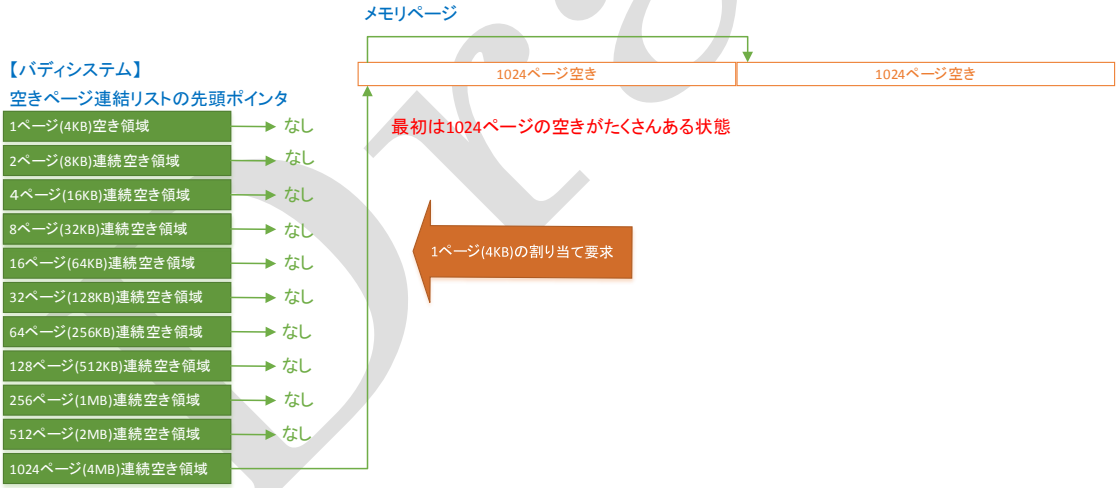
バディシステムは、効率的なページ管理ができるので、ほぼそのまま採用する。

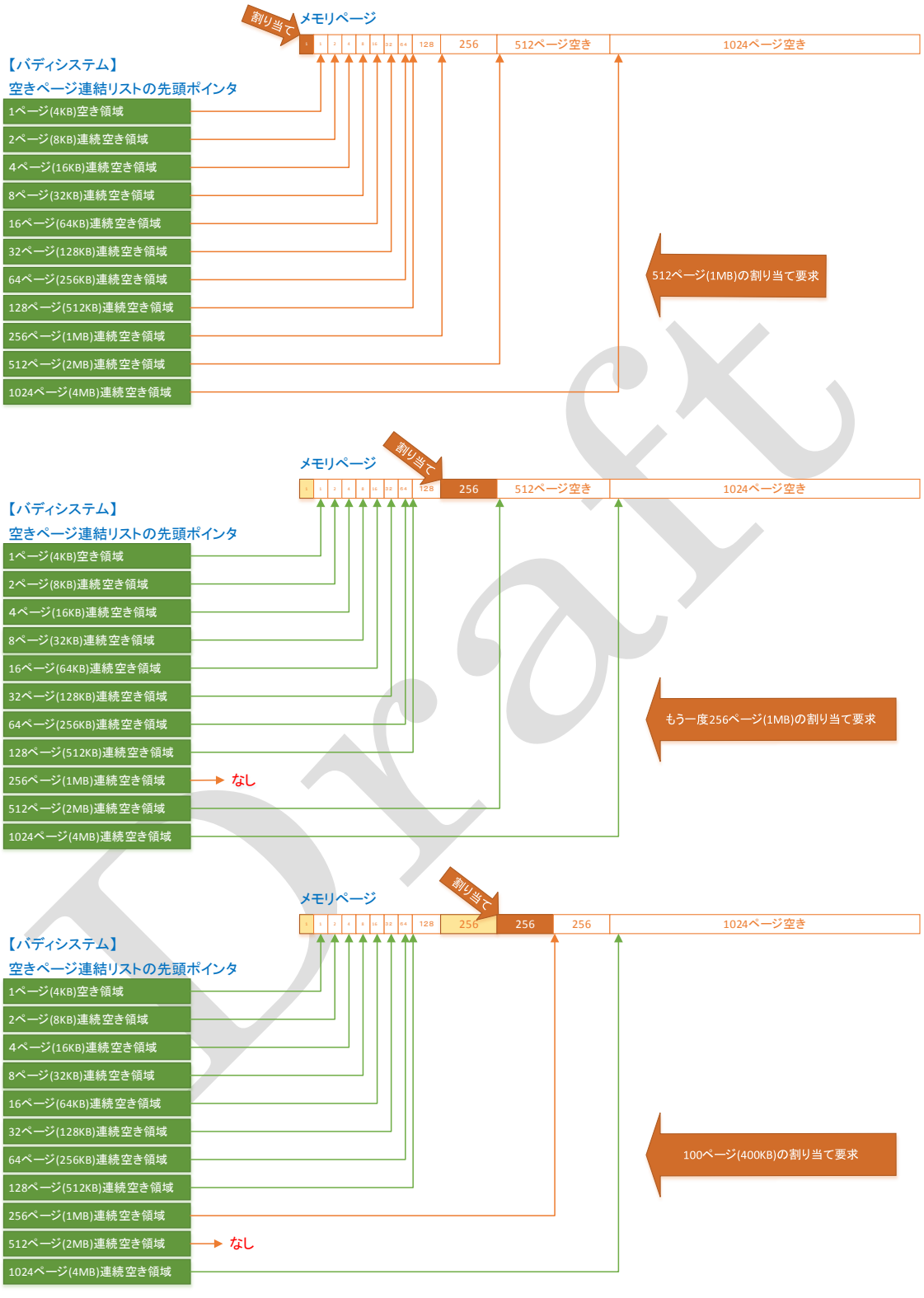
バディシステムのイメージ：

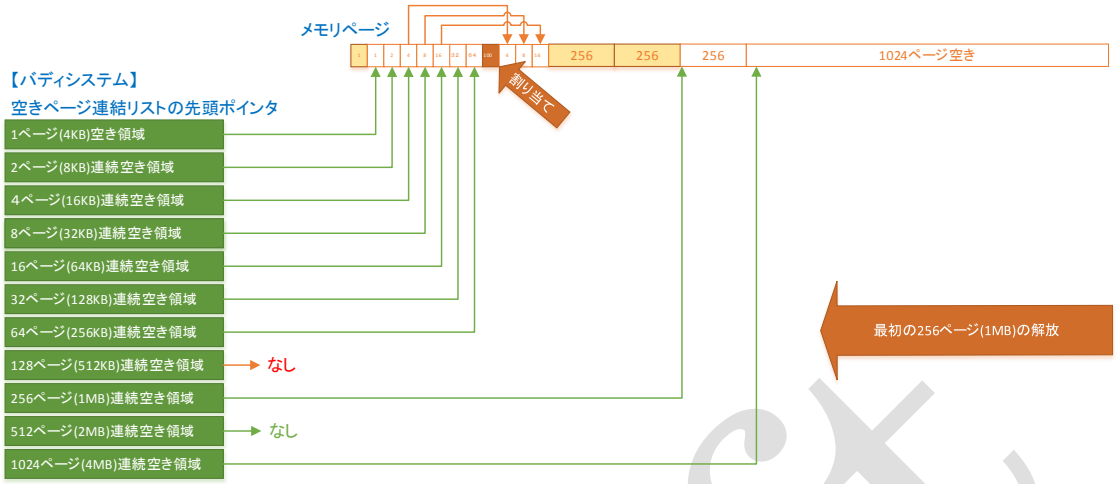


バディシステムは、11個の「連続空き領域（ページ）」連結リストを管理する

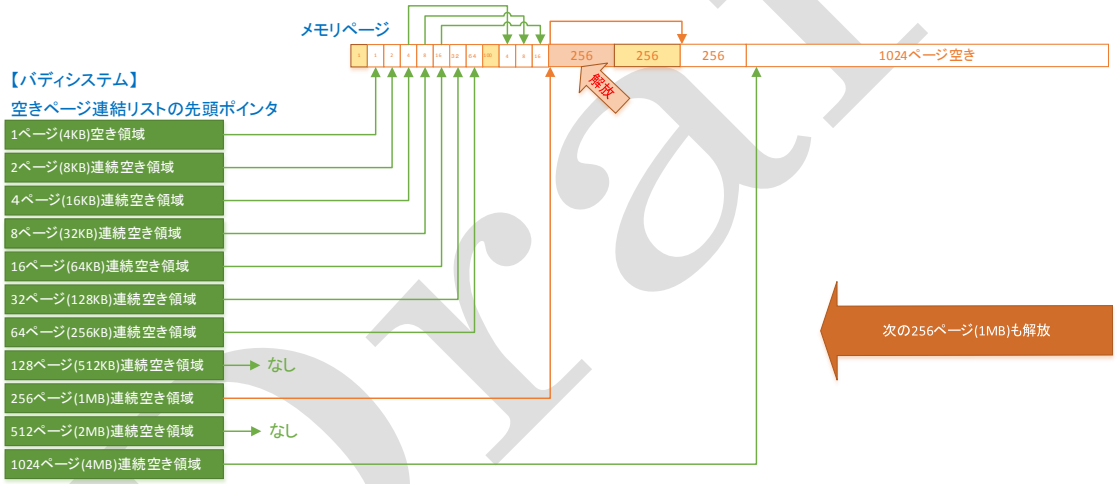
バディシステムのページ割り当てイメージ：

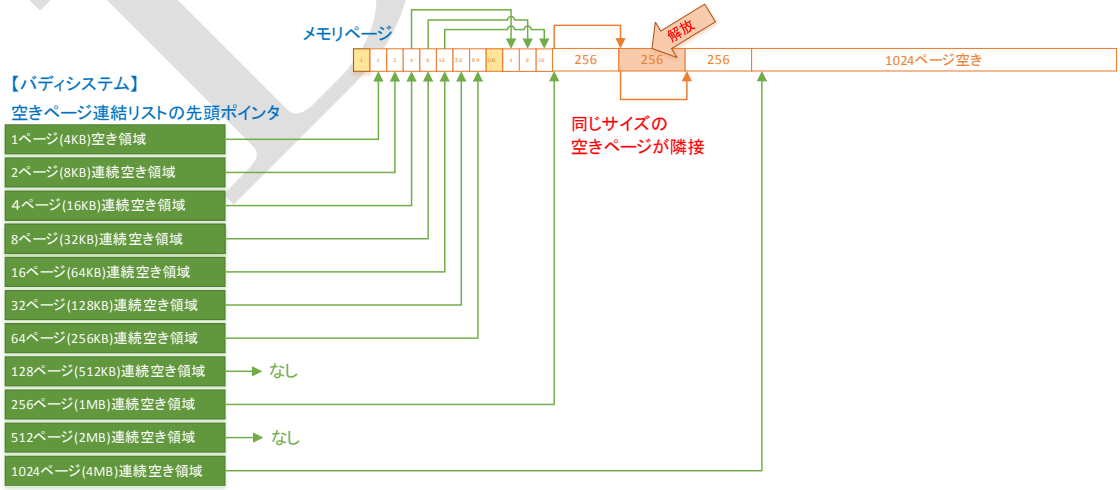


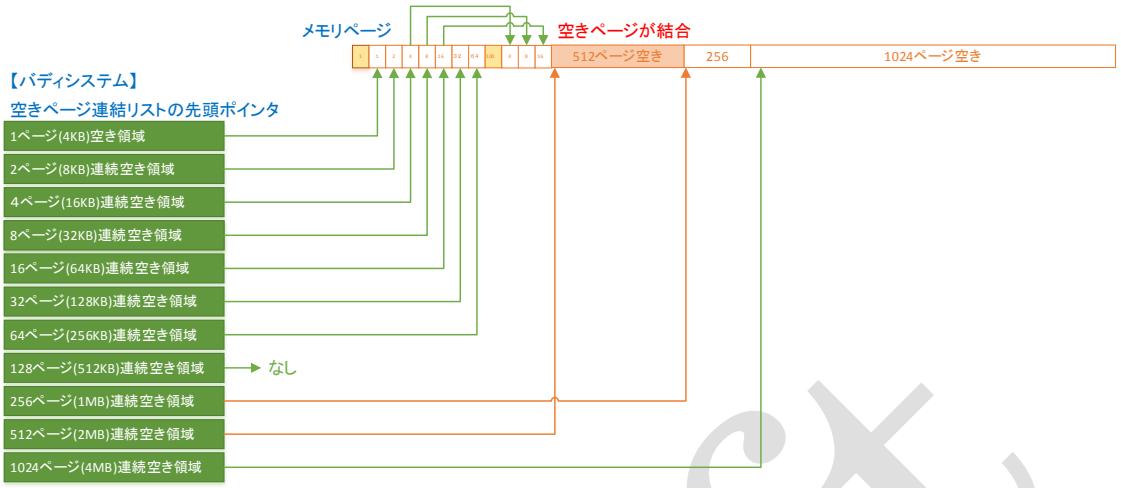

割り当て
メモリページ
128
256
512ページ空き
1024ページ空き
【バディシステム】
空きページ連結リストの先頭ポインタ
1ページ(4KB)空き領域
2ページ(8KB)連続空き領域
4ページ(16KB)連続空き領域
8ページ(32KB)連続空き領域
16ページ(64KB)連続空き領域
32ページ(128KB)連続空き領域
64ページ(256KB)連続空き領域
128ページ(512KB)連続空き領域
256ページ(1MB)連続空き領域
512ページ(2MB)連続空き領域
1024ページ(4MB)連続空き領域
512ページ(1MB)の割り当て要求
割り当て
メモリページ
128
256
512ページ空き
1024ページ空き
【バディシステム】
空きページ連結リストの先頭ポインタ
1ページ(4KB)空き領域
2ページ(8KB)連続空き領域
4ページ(16KB)連続空き領域
8ページ(32KB)連続空き領域
16ページ(64KB)連続空き領域
32ページ(128KB)連続空き領域
64ページ(256KB)連続空き領域
128ページ(512KB)連続空き領域
256ページ(1MB)連続空き領域
なし
512ページ(2MB)連続空き領域
1024ページ(4MB)連続空き領域
もう一度256ページ(1MB)の割り当て要求
割り当て
メモリページ
128
256
256
256
1024ページ空き
【バディシステム】
空きページ連結リストの先頭ポインタ
1ページ(4KB)空き領域
2ページ(8KB)連続空き領域
4ページ(16KB)連続空き領域
8ページ(32KB)連続空き領域
16ページ(64KB)連続空き領域
32ページ(128KB)連続空き領域
64ページ(256KB)連続空き領域
128ページ(512KB)連続空き領域
256ページ(1MB)連続空き領域
なし
512ページ(2MB)連続空き領域
1024ページ(4MB)連続空き領域
100ページ(400KB)の割り当て要求



バディシステムのページ解放イメージ：







【参考】Linux（x64）のバディシステム使用状況表示イメージ：

```
$ cat /proc/buddy/info          ← /proc/buddy/info に状況が常に記録されている
Node 0, zone      DMA      4      1      3      4      2      3      2      2      2      2      0
Node 0, zone    DMA32    685   1098    677    101     32     13     13     12     10      4      1
                  ↑x64 なのでゾーンが二つ
                        ↑2^0 ページから 2^10 ページまでの 11 種類の連続空き領域の個数が表示
                              （けっこう断片化していることがわかる）
```

Linux では、頻繁にページの割り当て（物理メモリのマッピング）と解放が繰り返されることがあるため、CPU のキャッシュ効率を上げる仕組みを実装しているが、そこまでは対応しない。

本システムが「ページ管理」を採用するのは、用途ごとのメモリの配分を実行時に調整できるようにするためであり、頻繁なページの開放は意図しない。

例えば、後述するプールアロケータの利用に際して、「8 バイトプール」の割り当て個数と「16 バイトプール」の割り当て個数のバランスを、事前決定しなくていいようにする、といったことが目的である。ページ単位で「8 バイトプールのページ」、「16 バイトプールのページ」のように扱い、要求が多いものは都度ページを追加して割り当てていく。

同様に、ヒープメモリの個々のメモリ管理（ノード）情報や、プールメモリを含む個々のメモリ割り当てのデバッグ情報なども、1 ページ単位で都度割りあてを行う。

本システムは、別途ヒープメモリを用意することもあり、2 ページ以上に渡る連続領域の確保が要求される機会がそこまで多くはないが、全くないわけではない。大きなサイズのメモリプールや、初期のメモリ配分、スラブアロケータの利用に際しては、やはりまとまったメモリが必要になる。

バディシステムは 2 ページ以上の連続空き領域を管理するためのものであるが、本システムにおいても有効活用できるものとなるため、採用する。

- **【採用】検討事項：プールアロケータ**

Python のメモリ管理では、256 バイト以下のメモリを無駄なく高速に管理するために、「低レベルメモリアロケータ」を実装している。

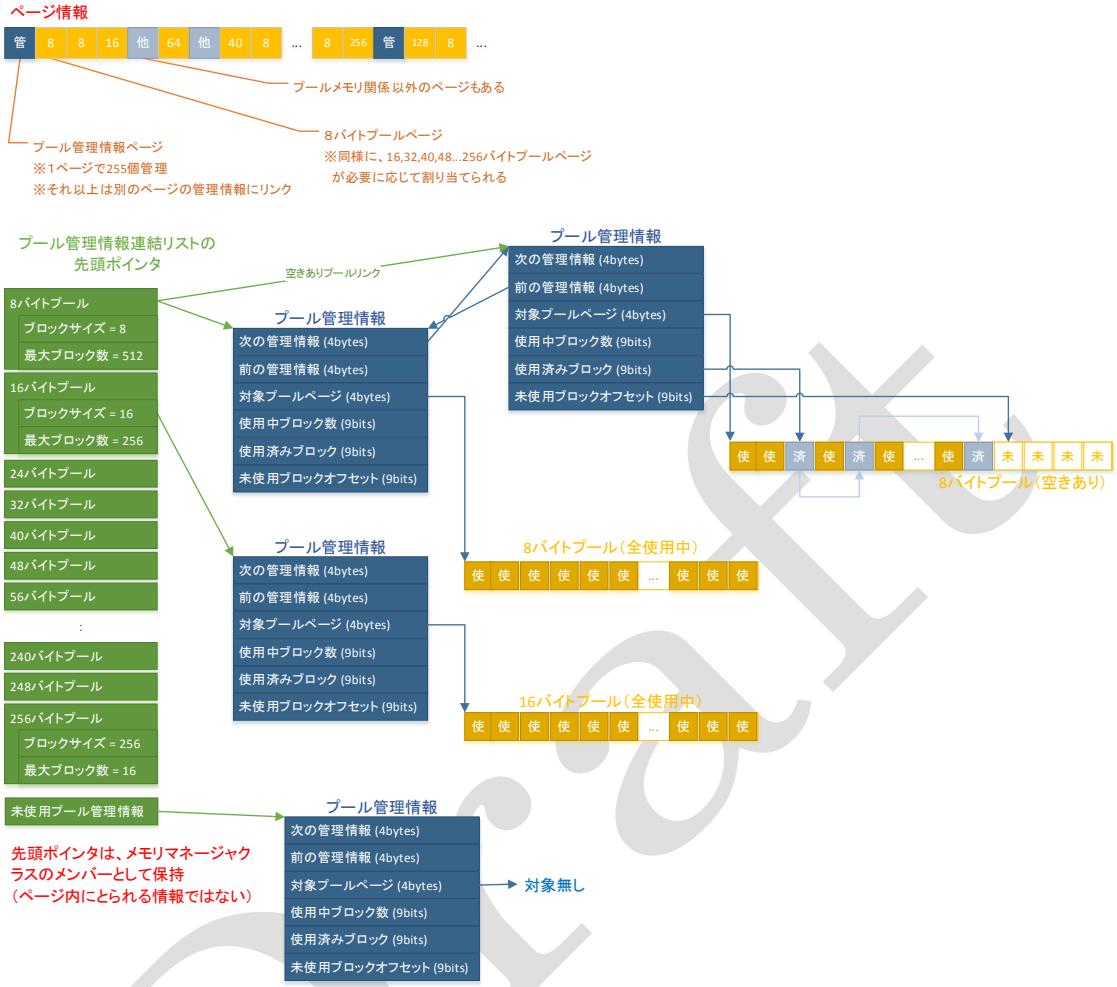
256 バイト以下の小さなメモリが、とりわけ大量にアロケートされる機会が多いことに基づく実装である。

4KB 単位の「プール」に区分けされたメモリを、メモリ確保の要求に応じて「8 バイト用（×512 ブロック）」、「16 バイト用（×256 ブロック）」、「24 バイト用（×170 ブロック）」...「256 バイト用（×16 ブロック）」のように割り当てて扱う。小さいメモリは多数の確保が予測されるため、最初のメモリ確保時点で 4KB 分まとめて予約する仕組みである。

例えば、8 バイト（以下）のメモリ確保要求の際に、512 個のメモリブロックが一気に用意され、以後同様に 8 バイト（以下）のメモリ確保要求があったらそのプールが使用される。プールを使い尽くした後にまた要求があったら、新たにプールを割り当てる。

「低レベルメモリアロケータ」は、ゲームプログラミングにおいても有効であるため、ほぼそのまま採用する。ただし、その呼称は、分かり易く「プールアロケータ」とする。

プールアロケータのイメージ：



4KB の「プール」は、そのまま 4KB の「ページ」に対応するため、必要に応じてページをプールに割り当てる方法を取る。

また、Python では、「アリーナ」と呼ばれる 256KB の領域をまとめて確保し、それを 4KB ずつ区切ったものを「プール」としている。256KB を超えるプールが必要になったらまたアリーナを確保する仕組みである。

本システムにおいては、アリーナはとくに扱わず、要求に応じてページを割り当てていくものとする。

ただし、ページ領域はプールアロケータ専用の領域ではなく、他の用途にも用いているため、制限を設ける。例えば「全ページ数の 80%」といった条件とする。それ以上はヒープからの割り当てを行う。

また、アリーナのようにまとめてプール用のメモリを確保せず、常に必要に応じて 1 ページずつプールに割り当てる。プール内のメモリがまるごと解放されたらページを解放し、他の用途に回せるようにする。

Python ではプールの状態をアリーナが管理しているが、本システムではアリーナを使用しないので、プールの状態を管理するための専用情報を設け、そのためのページを割り当てて扱う。

プール管理情報に 1 ページをまるごと割り当て、管理情報が 1 ページを超えるようならまた新たなページを管理情報に割り当てる。

なお、管理情報ページ内の管理情報がまるごと未使用状態になった場合、255 個の連結組み替えによって破棄することができるが、重くなるので実際に処理すべきかどうかは要検討。

プールの管理情報は、管理情報どうしを連結する双方向連結リストで構成し、対象プールのページ（アドレス）と使用中ブロック数、使用済みブロックのインデックス、未使用ブロックオフセットを扱う。1 情報あたり 16 バイトに収めると、4KB のページ内には 256 – 1 プールの管理情報を持つことができる。（ページの先頭にはページ自体の管理情報があるため -1 する）

仮に 1 ページまるごと 8 バイト用プールの管理情報だった場合、512 ブロック × 255 プール = 130,560 ブロックの 8 バイトメモリを管理できる。

- **【採用】検討事項：ヒープメモリ**

- **【採用】検討事項：Best-Fit アロケーションと合体（Coalesing）**

- **【採用】検討事項：メモリ再配置（Compaction）**

- **【採用】検討事項：スラブアロケータ（スラブキャッシュ）**

課題：スラブをどこに置くか？

▼ 処理効率の方針：高速なメモリ確保と解放のために

- **【採用】検討事項：ヒープメモリのチャンク連結**

- **【採用】検討事項：ヒープメモリの空きチャンク連結**

- **【採用】検討事項：ヒープメモリの空きチャンク連結**

▼ メモリ操作支援機能の方針：マルチスレッドと安全なメモリ操作のたまえに

ガベージコレクション、スマートポインタ

- **【不採用】検討事項： マークスウィープ GC**

- **【採用】検討事項： 参照カウンタ GC**

▼ 開発支援機能の方針：的確な問題追及のために

■■以上■■

■ 索引

索引項目が見つかりません。

ゲーム制御のためのメモリ管理方針

以　上