8章

メモリ管理

ページフレームの管理

・4KBページフレーム長をメモリ割り当ての標準単位として採用している理由
  ・ページフォルト例外が発生したとき、ページがあるけど使えないのか、ページが存在しないのか判断しやすい。
  ・メインメモリとディスク間のデータ転送は小さい単位の方が効率的。

ページディスクリプタ

- ページフレームの状態に関する情報は、ページ構造体として定義されたページディスクリプタに保持する。

_mapcount　このページフレームを参照しているページテーブルエントリの数

private　このページを使用するカーネル処理部が自由に利用するメンバ

mapping　このページをページキャッシュとして使う

index　複数のカーネル処理部ごとに異なった使い方をする

lru　ページの最長不使用順リストのポインタ

ページディスクリプタ

_count ページフレームの参照カウンタ

値が-1なら空きページフレーム

0以上なら1つ以上のプロセスに割り当てられているか、カーネルデータ用に使われている。

flags　ページフレームの状態を示す32個のフラグ。

フラグ名と意味は次スライド

表 8-2 ページフレームの状態を表すフラグ

フラグ名	意味
PG_locked	ページはロック状態。たとえば、ディスク I/O 操作で使っている
PG_error	このページの転送中に I/O エラーが発生した
PG_referenced	このページに最近アクセスがあった
PG_uptodate	ページの読み込みが完了したときに設定する。ただし、ディスク I/O エラーがあった場合を除く
PG_dirty	このページの内容を変更した(17 章の 17.3 を参照)
PG_lru	このページがアクティブページ LRU リスト、もしくは非アクティブページ LRU リスト内にある(17 章の 17.3.1 を参照)
PG_active	このページがアクティブページ LRU リスト内にある(17 章の 17.3.1 を参照)
PG_slab	スラブ用ページフレーム(8.2 を参照)
PG_highmem	ZONE_HIGHMEM ゾーン内のページフレーム(8.1.3 を参照)
PG_checked	Ext2 や Ext3 などのファイルシステムで使用するフラグ(18 章を参照)
PG_arch_1	80x86 アーキテクチャでは未使用
PG_reserved	カーネルコードが予約している、もしくは使うことができないページフレーム
PG_private	ページディスクリプタの private メンバを使用している
PG_writeback	writepage メソッドを用いてこのページをディスクに書き込んでいる(16 章を参照)
PG_nosave	サスペンドやレジュームで使用するフラグ
PG_compound	拡張ページング機構を通して使用しているページフレーム(2 章の 2.4.2 を参照)
PG_swapcache	スワップキャッシュ用ページ(17 章の 17.4.6 を参照)
PG_mappedtodisk	ディスク上のブロックに対応するデータを格納しているページフレーム
PG_reclaim	メモリ回収のためにこのページをディスクへ書き出すことを示す
PG_nosave_free	サスペンドやレジュームで使用するフラグ

不均等メモリアクセスアーキテクチャ（NUMAアーキテクチャ）

- CPUからの距離が異なるメモリへのアクセス時間は異なる。

↓

- カーネルは、時間のかかるアクセスの回数を減らす

↓

- カーネルは、対象のCPUがよく参照するデータを保存する場所を最適化する（アクセス時間が短い)

- ノードディスクリプタの表は次スライド

表 8-3　ノードディスクリプタのメンバ

型	名前	説明
struct zone []	node_zones	このノード用のゾーンディスクリプタの配列
struct zonelist []	node_zonelists	ページ割り当てに使う zonelist 構造体の配列(8.1.3 を参照)
int	nr_zones	ノードにあるゾーン数
struct page *	node_mem_map	ノードのページディスクリプタの配列
struct bootmem_data *	bdata	カーネルの初期化時に使用
unsigned long	node_start_pfn	ノードの先頭ページフレームのインデックス
unsigned long	node_present_pages	メモリホールを除いたページフレーム単位のメモリノードの大きさ
unsigned long	node_spanned_pages	メモリホールを含んだページフレーム単位のメモリノードの大きさ
int	node_id	ノードの識別子
pg_data_t *	pgdat_next	メモリノードのリストの次の要素
wait_queue_head_t	kswapd_wait	kswapd ページアウトデーモンが使う待ちキュー(17 章の 17.3.4 を参照)
struct task_struct *	kswapd	kswapd カーネルスレッドのプロセスディスクリプタへのポインタ
int	kswapd_max_order	kswapd が回収するブロック数(2 を底とする対数で表記)

メモリゾーン

Linuxにおけるメモリレイアウト

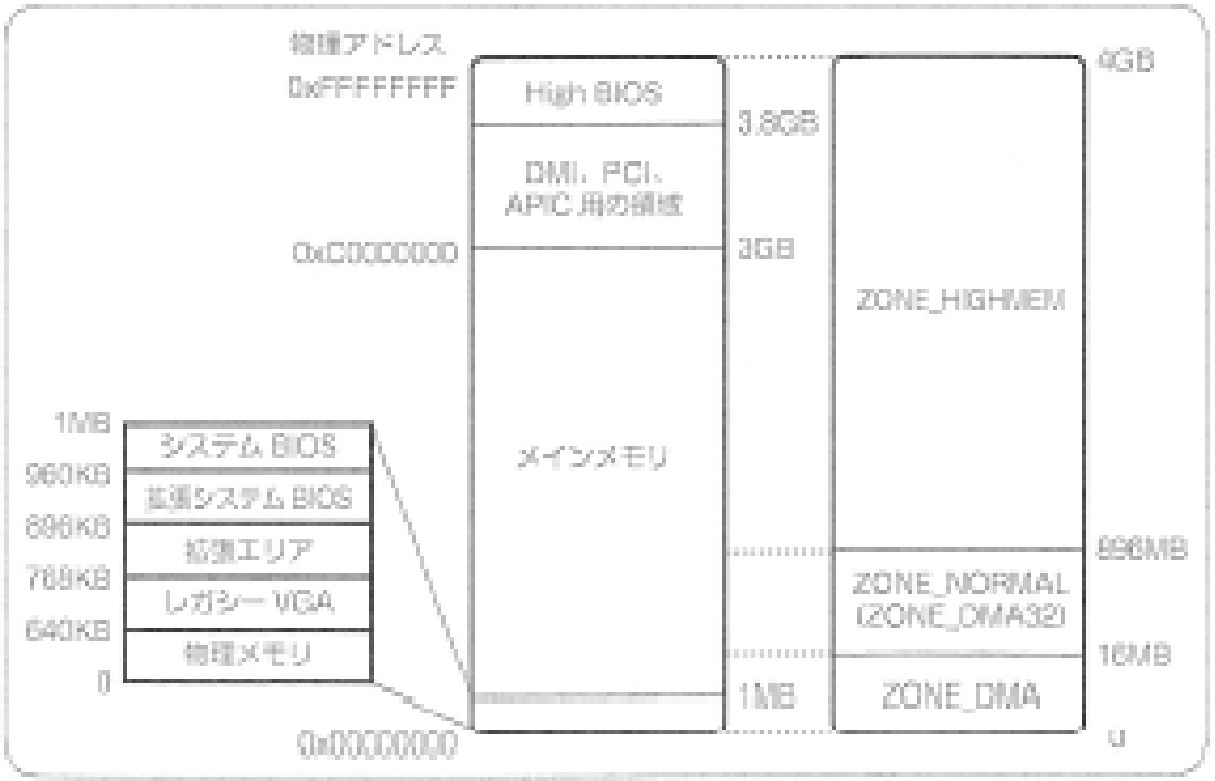
IA-32用のLinuxカーネルは、物理アドレス空間をゾーンという3つの領域に分割し、名前を付けています(図2の右側)。物理アドレス空間をゾーンに分割する理由は、ゾーンごとに、大きさや使われ方に特徴があるためです。

先頭0～16Mバイトの領域をZONE_DMAといいます。DMAはDirect Memory Accessの略で、デバイスがCPUを介さずに物理メモリにアクセスする方式のことです。この領域は16Mバイト以下の物理メモリにしかアクセスできないレガシーなデバイス(具体的にはISAバスに接続されるデバイス)がDMA転送を行うために提供されています。

16～896Mバイト領域をZONE_NORMALといいます。Linuxは、この領域にリソース管理に必要な情報を保持します。

896Mバイト以降の領域を、ZONE_HIGHMEMといいます。

●図2　Intel G35チップセットの物理メモリのレイアウトとそれに対応するLinuxのゾーン



空きページフレームの予約

- 空きメモリが十分なら、カーネルはメモリ割り当て要求を即座に処理。
- 十分でなければ、カーネルはメモリ回収をする。メモリがある程度開放されるまで割り当て要求の処理を中断。
- 割り込み処理等、中断できないメモリ割り当て要求もある。中断できず、空きメモリが足りないと、要求が失敗してしまう。失敗の可能性を減らすため、カーネルはページフレームを予約しておく。

ゾーンごとのページフレームアロケータ

メモリアロケータ

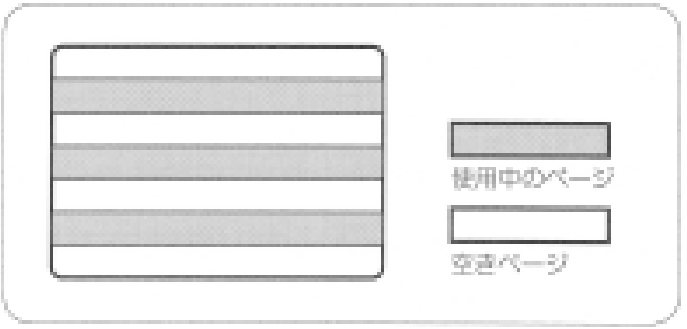
メモリの断片化の問題

メモリの断片化とは、割り当てを細かく行った結果、メモリの割り当てが虫食いにあったような状態になることを指します。図5では、空いているページが4つありますから、合計12Kバイトのメモリが利用可能はなずです。ところが、12Kバイトの「連続した」物理メモリは割り当てることができません。仮想メモリを使用しているのにもかかわらず、連続した物理メモリを割り当てる必要がある状況があります。デバイスとメモリ間でデータをCPUを介さずに転送するDirect Memory Access(DMA)のようなしくみを用いる場合が、代表的な例です。なぜなら、DMA要求を受け付けているデバイスは、今CPUがどのページテーブルを使用しているかわからないため、直接物理メモリに対して転送を行う必要があるからです。

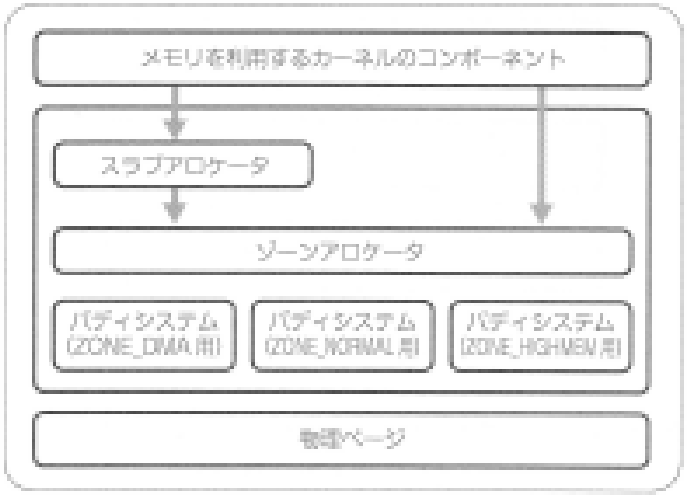
こういった要求から、Linuxカーネルは3つのメモリアロケータ(「バディシステム」「ゾーンアロケータ」「スラブアロケータ」)を用いてメモリを管理しています。これらの関係を図6に示します。

ゾーンアロケータは、バディシステムのフロントエンドです。最初に説明したように、Linuxでは物理メモリをゾーンという領域に分割しています。各ゾーンにはメモリ量の大小や使われ方に特徴があるため、ZONE_DMA、ZONE_ NORMAL、ZONE_ HIGHMEMの3つの領域に分けて管理します。

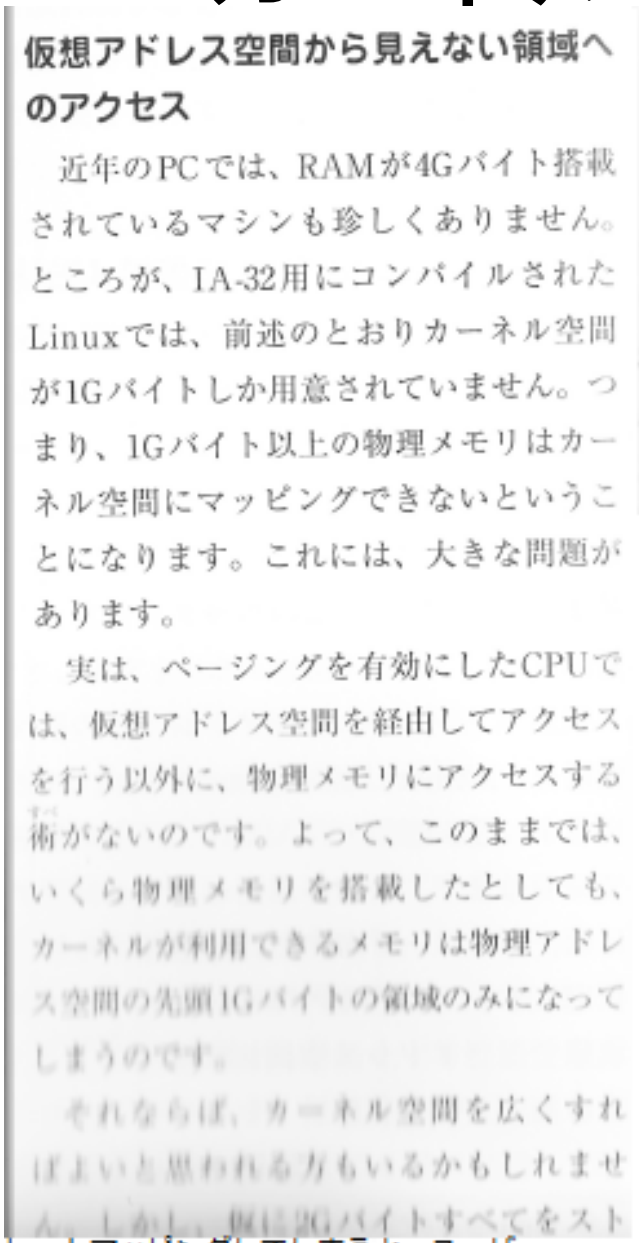
● 図5　虫食い状態の物理メモリ



● 図6　Linuxにおけるメモリアロケータの関係



高位メモリのページフレームのカーネルマッピング



仮想アドレス空間から見えない領域へのアクセス

近年のPCでは、RAMが4Gバイト搭載されているマシンも珍しくありません。ところが、IA-32用にコンパイルされたLinuxでは、前述のとおりカーネル空間が1Gバイトしか用意されていません。つまり、1Gバイト以上の物理メモリはカーネル空間にマッピングできないということになります。これには、大きな問題があります。

実は、ページングを有効にしたCPUでは、仮想アドレス空間を経由してアクセスを行う以外に、物理メモリにアクセスする術がないのです。よって、このままでは、いくら物理メモリを搭載したとしても、カーネルが利用できるメモリは物理アドレス空間の先頭1Gバイトの領域のみになってしまうのです。

それならば、カーネル空間を広くすればよいと思われる方もいるかもしれません。しかし、仮に2Gバイトすべてをストレートマッピングしてしまうと、ユーザプロセスの使用できる仮想アドレス空間が2Gバイトに減ってしまい、実質ユーザプロセスが使えるメモリがなくなってしまいます。そこで、IA-32用のLinuxでは、仮想アドレス空間のうち、最上位128Mバイトの領域を、一時的なマッピングを行うために予約しています。そして、HIGHMEM領域にアクセスする際は、この領域のいずれかに一旦マップを行ったうえで、マップした仮想アドレスに対してアクセスを行います。

一時的カーネルマッピングだけでなく、
・永続的カーネルマッピング
・非連続メモリ割り当て
の3種類の方法がある。
ただし何をやっても、高位メモリのマッピングには128MBまでしか使えない。

バディシステム

バディシステム

バディシステムは、すべての空き物理ページを2の階乗(1、2、4、8、…)個のブロックに分類し、空きブロックの大きさと割り当て要求を比較しながら物理ページの割り当てを行います[※4]。メモリの割り当て要求がくると、バディシステムは空きブロックの中で、

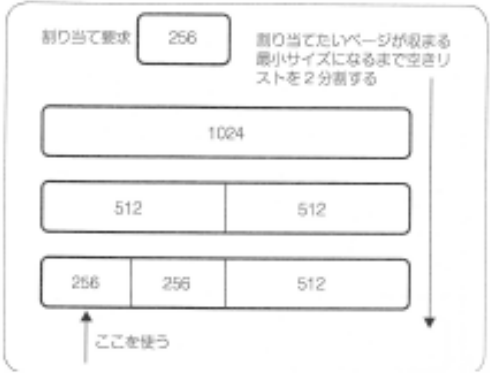
(1)要求サイズが収まる
(2)最も小さい

という2つの条件を満たす空きブロックが空いているかチェックします。空いていればこのブロックを割り当て、空いていなければ今見ているブロックの2倍の大きさの空きブロックを見にいきます。ここでメモリが見つかった場合は、要求サイズが収まる最も小さいブロックにまで空きブロックを分割したうえで、分割したメモリに要求されている大きさのメモリを割り当てます。一方、最大サイズの空きブロックまで見にいって、空きブロックがないと判断された場合、バディシステムはエラーを返して処理を終了します。

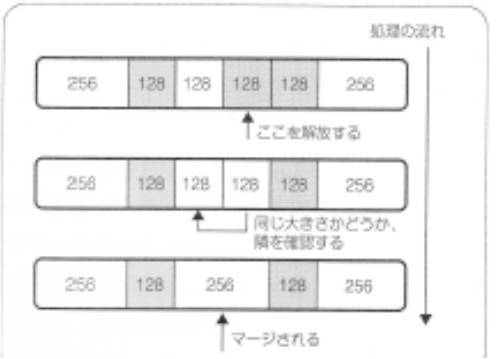
具体例として、1024個の物理ページ(合計4Mバイト)に対して、255個の連続した物理ページ(1Mバイトのメモリ)のメモリ獲得要求があった場合のことを考えます(**図7**)。この場合、2の階乗で255より大きい最小の数は256になるので、256個用の空きブロックがあるかどうかを探索します。ある場合はそのまま割り当てを行います。なければ、それより大きな空きブロック、この場合は512個用の空きブロックがあるかどうかをチェックします。512個用の空きブロックがないので、1024個の空きブロックを探索します。1024個の空きブロックが見つかったので、バディシステムは1024個の空きブロックを512個の空きリスト2つに分割します。これでもまだ割り当て要求サイズより大きいので、さらに512個用の空きブロックを256個用の空きブロック2つに分割し、そのうちの1つを割り当てます。割り当てたブロックは、使用中リストを用いてたどれるようにしておきます。

逆に、メモリ解放時には、隣り合う物理メモリが同じ大きさであれば、2倍の大きさの1つのブロックにマージされます(**図8**)。こうすることで、より大きな連続した物理ページの確保に対応できます。

◆図7　バディシステムによるメモリの割り当て



◆図8　バディシステムによるメモリ解放



CPUごとのページフレームキャッシュ

- カーネルは頻繁に1ページフレームだけのメモリ割り当て要求とメモリ開放を行う。

↓

- これに対する性能を稼ぐため、各メモリゾーンはCPUごとのページフレームキャッシュを持っている。

↑

活性キャッシュ

不活性キャッシュがある。

CPUごとのページフレームキャッシュ

- ページフレーム割り当て直後にカーネルやユーザ空間のプロセスがデータを書き込む場合、活性キャッシュのページフレームを使うと速くなる。
- これと対照に、不活性キャッシュのページフレームはDMA操作によってデータを書き込むのに適する。

ゾーンアロケータ

- バディシステムのフロントエンド
- 各メモリゾーンの中から、メモリ割り当て要求を満たす大きさの空きページフレームを持つメモリゾーンを特定する。

↓処理が大変な理由

- 予約ページフレームをなるべく使わないようにする
- カレントプロセスを中断できるならページフレームの回収処理をしなければならない。
- なるべく小さいメモリゾーンの使用は避ける

メモリ領域の管理

スラブアロケータ

バディシステムのみを用いてメモリ確保を行う場合、たった1バイトのメモリ確保を10回行う場合についても、10ページ、すなわち40Kバイト分だけメモリを確保する必要があります。これは明らかに無駄です。

Linuxでは、より効率良くメモリを管理するため、スラブアロケータを採用しています。スラブアロケータは、バディシステムから得た連続ページを、より細かい単位で管理します。

図9に、スラブアロケータの概要を示します。スラブアロケータは、バディシステムから確保したメモリをキャッシュ[85]と呼ばれる「同じ種類のオブジェクトの集合」に分割します。ここでいう同じ種類のオブジェクトとは、同じ大きさを持つ構造体のインスタンスのことを指します。1つのキャッシュには、1つの種類の構造体を登録することができます。別の構造体をスラブを用いて管理する場合は、新たにキャッシュを作成する必要があります。

キャッシュとスラブの関係

キャッシュを管理するための構造体を、キャッシュディスクリプタといいます。カーネルがすべてのキャッシュを把握するために、すべてのキャッシュディスクリプタは、リストでつながっています。

キャッシュは、バディシステムから確保された連続した物理ページの集まりです。スラブアロケータは、この物理ページの集まりに対して、同じ大きさのオブジェクトを割り当てていきます。オブジェクトの入れ物をスラブといいます。

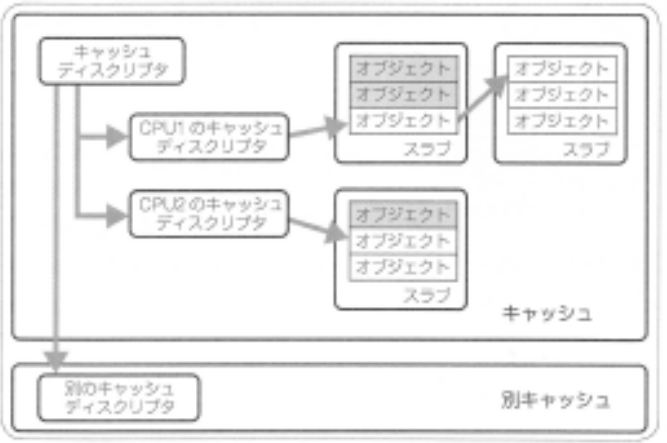
CPUが複数存在するシステムでも性能向上が見込めるように、スラブアロケータはCPUごとにキャッシュを保持し、メモリ管理を行います。これにより、CPU間のロックを行うことなく、高速にメモリを割り当てることができます。CPUごとのキャッシュを管理する構造体を、CPUキャッシュディスクリプタと呼ぶことにします。

CPUキャッシュディスクリプタは、次の割り当てに使う空きオブジェクトへのポインタを保持しており、割り当て要求がきた際に、このオブジェクトを使用します。

スラブ内に保持されている各オブジェクトは、次の空きオブジェクトへのポインタを保持しています。オブジェクトの割り当て時、CPUキャッシュディスクリプタはこのポインタを参照して、オブジェクトをどのスラブから利用すればよいかを決定します。

空きオブジェクトを持つスラブがなくなると、スラブアロケータはバディシステムに対して新たにメモリ割り当て要求を行います。この操作により、空きオブジェクトのみのスラブがキャッシュに追加されることになります。

• 図9　スラブアロケータのイメージ

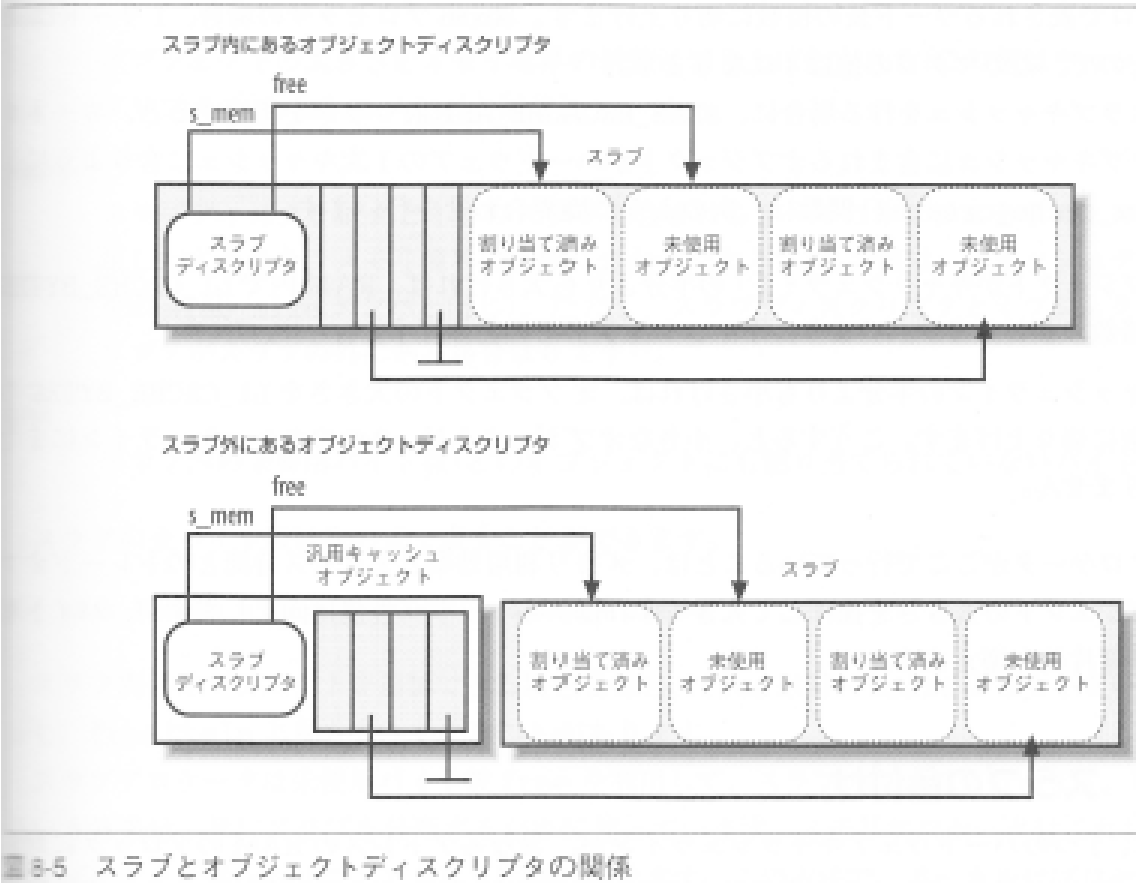


メモリ領域の管理

- カーネルがスラブを削除する条件
  - スラブキャッシュの空きオブジェクトが多すぎる場合
  - 定期的なタイマ関数によって、空きオブジェクトだけ持つスラブを解放する場合

オブジェクトディスクリプタ

- オブジェクトディスクリプタはスラブディスクリプタの直後に存在(下図)
- 次の空きオブジェクトのインデックスを保持



(上段)内部オブジェクトディスクリプタ
スラブ内のオブジェクトの直前に配置

(下段)スラブ外の汎用キャッシュ内に存在。これを格納するメモリ領域の大きさはスラブ内のオブジェクト数で変化。

図8-5　スラブとオブジェクトディスクリプタの関係

メモリ上でのオブジェクトの境界合わせ

- カーネルは、オブジェクトの先頭の物理アドレスを2のべき乗に切り上げる（アライメント：境界合わせ）
- プロセッサがメモリにアクセスする場合、その物理アドレスがワード長（メモリバスの幅）の倍数に合っていると、早くアクセスできる。
- オブジェクトがL2キャッシュラインの半分より大きければ、キャッシュラインの先頭に境界合わせする。
- 半分より小さければ、キャッシュラインの約数に切り上げ、小さなオブジェクトを2つのキャッシュラインにまたがらせない。
- これは効率と時間のトレードオフである。

  （オブジェクトを大きくすればキャッシュ性能は向上するが、メモリ断片化も増える）

スラブの色付け

- 違うスラブ内の同じオフセットにあるオブジェクトは、比較的高い確率で同じキャッシュラインにマッピングされる。

↓

- RAMの違う場所にあるオブジェクト同士が同じキャッシュラインを奪い合うことで無駄なメモリサイクルが発生。

↓

- 「スラブの色付け」で無駄を減らす
  - スラブ内の未使用バイト数に基づいて色付けする。
  - メモリアロケータは色に基づいて、オブジェクトを異なるリニアアドレスに分散させる。

空きスラブオブジェクトの ローカルキャッシュ

- プロセッサ間のスピンロックを減らし、ハードウェアのキャッシュを有効利用するために、スラブアロケータはCPUごとにスラブローカルキャッシュを持つ。
- スラブの割り当てと解放の大半はスラブローカルキャッシュに対して行う。
- スラブキャッシュを作るとき、kmem_cache_create関数は、ローカルキャッシュの大きさを決める。この大きさに従い、CPUごとのローカルキャッシュを確保し、キャッシュディスクリプタのarrayメンバが各ローカルキャッシュを指すようにする。ローカルキャッシュの大きさは最小で1、最大で120。

スラブオブジェクトの割り当て・解放

- P.359～363を読んでください^^;

汎用オブジェクト

- それほど頻繁に行われないメモリ領域の要求は、汎用キャッシュが処理する。
- 汎用キャッシュには32～131072バイトまでのサイズのオブジェクトがある。
- 汎用キャッシュのオブジェクトを獲得：kmalloc関数
- kmallocで割り当てられたオブジェクトの解放：kfree関数

メモリプール

- メモリ不足状態のとき、動的メモリを確保するために使う。
- １つのメモリプールと1つのカーネル処理部は１対１対応。
- スライド９で説明した、予約したページフレームを使っても間に合わないような場合だけ使う、非常手段。
- メモリプールは、スラブオブジェクトを蓄えるため、スラブアロケータを利用して作ることが多いが、メモリプール自体はどんな種類の動的メモリも扱える。

メモリプール

- メモリプールメモリ切片を割り当てるため、カーネルはmempool_alloc関数を呼び出す。
- メモリ切片の割り当てに成功すれば、この関数はメモリプールに触らず、確保したメモリ切片を返す。
- 失敗したときは、メモリプール内のメモリ切片を返す。
- メモリプールを使い果たしたときは、メモリプールにメモリ切片が解放されるまで、カレントプロセスの実行を止める。

非連続メモリ領域の管理

- 普通は、メモリ領域を連続したページフレームにマッピングするほうが、キャッシュの効率が良く、メモリアクセス回数を抑えられるので、良い。
- ただし、メモリ領域への要求がそれほど頻繁でないなら、連続したリニアアドレスを持つ非連続ページフレームに割り当てる方が良いこともある。
  - 利点：ページ単位のメモリ断片化を避けられる
  - 欠点：カーネルページテーブルを細かく操作する必要がある