日曜研究室

技術的な観点から日常を綴ります

[xv6 #14] Chapter 1 – The first process – Code: Physical memory allocator

テキストの22〜23ページ

本文

アロケータのデータ構造は、割り当て可能な物理メモリページの空きリストである。
空きページのリストの要素はrun構造体（struct run）である。
アロケータはその構造体を保持するためのメモリをどこに確保するか？
空きページそれ自身にrun構造体は置かれる。（置けなくなるまで）
空きリストはスピンロック（spin lock）で保護される。
空きリストとロック情報は、その構造体のフィールドをロックで保護することを確実にするために、構造体のなかに一緒にまとめられている。
今のところは、そのロック機構とacquire関数・release関数は無視しなさい。
それについては第3章で詳細を説明する。
（ロックについてはspinlock.h, spinlock.cで実装されている。spinlock構造体やacquire関数・release関数はそこで定義されています。本文に従いここではまだそのソースは掲載しません。）
（以下は、kalloc.cで定義されているrun構造体です。）

```
1 struct run {
2   struct run *next;
3 };
```

main関数は、アロケータを初期化するためにkinit関数を呼ぶ。
kinitは、まずどれだけの物理メモリが利用可能か決定しなければならない。
しかし、それはx86では難しい。
その代わり、マシンは少なくとも240MB（PHYSTOP）の物理メモリを持つと仮定し、そして空きメモリの最初のプールとしてカーネルの終端からPHYSTOPの間の全てのメモリを使う。
kinit関数は、カーネルの終端からPHYSTOPの間のメモリの全てのページのアドレスとともにkfree関数を呼ぶ。
kfree、アロケータの空きリストにそれらのページ全てを追加する。
アロケータはメモリ無しで開始する。

kfreeはアロケータにいくらか管理すべきものを与える。
（かなり訳があやふや。原文: these calls to kfree give it some to manage.）
（以下は、kalloc.cのkinit, kfree関数）

```
01 // Initialize free list of physical pages.
02 void
03 kinit(void)
04 {
05   char *p;
06
07   initlock(&kmem.lock, "kmem");
08   p = (char*)PGROUNDUP((uint)newend);
09   for(; p + PGSIZE <= (char*)p2v(PHYSTOP); p += PGSIZE)
10     kfree(p);
11 }
```

```
01 //PAGEBREAK: 21
02 // Free the page of physical memory pointed at by v,
03 // which normally should have been returned by a
04 // call to kalloc().  (The exception is when
05 // initializing the allocator; see kinit above.)
06 void
07 kfree(char *v)
08 {
09   struct run *r;
10
11   if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)
12     panic("kfree");
13
14   // Fill with junk to catch dangling refs.
15   memset(v, 1, PGSIZE);
16
17   acquire(&kmem.lock);
18   r = (struct run*)v;
19   r->next = kmem.freelist;
20   kmem.freelist = r;
21   release(&kmem.lock);
22 }
```

アロケータは、上位メモリにマップされた仮想アドレスによって物理ページを指す。
指すのは物理ページであって物理アドレスではない。
それが、kinitが、PHYSTOP（の物理アドレス）を変換するためにp2v(PHYSTOP)を使う理由である。
アロケータはたまに、アドレス関連の計算を行うために、アドレスを整数として扱う。
（例えば、kinitで全てのページを走査してる部分）
そして、その他はメモリを読み書きするためにアドレスをポインタとして使う。
（ページごとに保持されたrun構造体を操作する部分）
このアドレスの2つの使い方が、アロケータのコードがC言語のキャストでいっぱいである、主な理由である。
他の理由としては、解放や割り当てが本質的にメモリのタイプを変えることが挙げられる。

kfree関数は、渡されたページに対応するメモリを解放して1で埋めることから始まる。
これは、解放したあとコードがそのメモリを使うと、以前の正しい内容の代わりにゴミを読み込んで

しまう、という事態を引き起こすだろう。
うまくいけば、そんなコードは速やかに中断するだろう。
次に、kfree関数は変数vをrun構造体のポインタにキャストし、空きリストの始まりをr->nextに記録し、空きリストにrを代入する。
kalloc関数は、空きリストの最初の要素を返し、それをリストから削除する。
（以下は、kalloc.cのkalloc関数）

```
01 // Allocate one 4096-byte page of physical memory.
02 // Returns a pointer that the kernel can use.
03 // Returns 0 if the memory cannot be allocated.
04 char*
05 kalloc(void)
06 {
07   struct run *r;
08
09   acquire(&kmem.lock);
10   r = kmem.freelist;
11   if(r)
12     kmem.freelist = r->next;
13   release(&kmem.lock);
14   return (char*)r;
15 }
```

カーネルの最初のページテーブルを作成するとき、setupkvmとwalkpgdirは、kallocの代わりにenter_allocを使う。
このメモリアロケータはカーネルの終わりを1ページ分動かす。
enter_alloc関数は、カーネルのデータセグメントの終わりのすぐ次のアドレスを指すendというシンボルを使う。
（endは手打ちじゃなくてリンカが設定するらしい？）
PTEは4096バイト境界で区切られた物理アドレスを指すことしかできない。
で、enter_alloc関数は、その境界で区切られた物理アドレスだけを割り当てる事を確実にするためにPGROUNDUPを使う。
（mmu.hに#define PGROUNDUP(sz) (((sz)+PGSIZE-1) & ~(PGSIZE-1))と定義されています。これによってどんなアドレスを渡してもPGSIZE=4096バイトで分割された領域の最初のアドレスが返ってきます。）
enter_allocで割り当てられたメモリはずっと解放されない。
（以下は、kalloc.cのenter_alloc関数）

```
01 // A simple page allocator to get off the ground during entry
02 char *
03 enter_alloc(void)
04 {
05   if (newend == 0)
06     newend = end;
07
08   if ((uint) newend >= KERNBASE + 0x400000)
09     panic("only first 4Mbyte are mapped during entry");
10   void *p = (void*)PGROUNDUP((uint)newend);
11   memset(p, 0, PGSIZE);
12   newend = newend + PGSIZE;
13   return p;
14 }
```

感想

run構造体おもしろいですね。
通常リンクリストを使うときは、次の要素へのポインタに加え保持するデータそのもの（もしくはデータへのポインタ）を持つわけです。
しかし、run構造体は、次の要素へのポインタ（32bit）を実メモリ上で専有する事自体がすでに用をなしてるので
データを持たなくていいってことなんでしょうね。

前回ちらっと言及されてましたが、通常使うアロケータはkalloc、ブート時に使うとりあえずのアロケータはenter_allocって事みたいですね。

通常使うアロケータは空きページを数珠つなぎに記録したリンクリスト（空きリスト）を使って、メモリの割当時にその空きリストから割り出し、メモリの開放時にその空きリストに戻すということをやってるみたいです。

今まで、git clone git://pdos.csail.mit.edu/xv6/xv6.git でクローンしたローカルのリポジトリのソースをそのまま読んでて、以前にもソースとテキストで不整合な部分があるみたいな事を書いたかと思いますが、今回その不整合がかなり大きいのでおかしいなと思い調べたら、テキストと一緒に読み進めるときはxv6-rev6というタグがついたコミットを使わなければいけないみたいだということが判明しました。
今見たらオフィシャルサイト http://pdos.csail.mit.edu/6.828/2011/xv6.html にも書いてありますね。
クローンしてからgit checkout -b xv6-rev6 xv6-rev6とすると作業コピーがテキスト準拠版になるみたいです。

前回までに、テキストの方を読み替えて対応してた部分がいくつかあったかと思います。
気が向いたら修正しますが、理解の妨げになるほどの差異は無かったかと思いますので、とりあえずそのままで今回からテキスト準拠版のソースを使っていきます。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/2/19 日曜日 [http://peta.okechan.net/blog/archives/1282] |