日曜研究室

技術的な観点から日常を綴ります

[xv6 #19] Chapter 1 – The first process – Real world

テキストの29〜30ページ

本文

多くのオペレーティングシステムは、そのプロセスのコンセプトを採用され、大部分はxv6のプロセスと類似している。
実際のOSの世界では、allocprocで線形時間検索をする代わりに、明示的な空きリストを使って一定の時間で空いているproc構造体を探す。
xv6は分かりやすさを優先し線形探索を使っている。（他も最初はだいだいそうだ）

大部分のOSがそうであるように、xv6もマッピングと保護のためにページングハードウェアを利用する。
大部分のOSでは、xv6よりページングの使い方が洗練されている。
例えば、xv6はディスクへのページング要求やfork時のcopy-on-writeや共有メモリやスタックの動的拡張などの機能を欠いている。
（ディスクへのページング要求がない＝いわゆるディスクスワップの機能がない。あとfork時にCOWしないということはforkの瞬間にメモリを割り当てるためメモリ効率やパフォーマンスが劣ることになる。）
x86もまたセグメントを使ったアドレス変換をサポートするが、xv6は違うCPUで違う値を持ちながら同じ固定アドレスにあるprocのように、CPUごとの値を持つことを実装するためのお決まりのトリックのためだけに使う。（seginit参照）
セグメントに対応してないアーキテクチャにおけるCPUごと（もしくはスレッドごと）のストレージの実装では、CPUごとのデータ領域を指すポインタを保持するためにレジスタを専有するだろう。
セグメントの機能を使う事は追加の労力を必要とするが、x86はそのように使えるいくつかの汎用的なレジスタを持っていて、それに労力を費やす価値はある。

vm.cのseginit関数

```
01 // Set up CPU's kernel segment descriptors.
```

```
02 // Run once on entry on each CPU.
03 void
04 seginit(void)
05 {
06   struct cpu *c;
07
08   // Map "logical" addresses to virtual addresses using identity
   map.
09   // Cannot share a CODE descriptor for both kernel and user
10   // because it would have to have DPL_USR, but the CPU forbids
11   // an interrupt from CPL=0 to DPL=3.
12   c = &cpus[cpunum()];
13   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
14   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
15   c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
16   c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
17
18   // Map cpu, and curproc
19   c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
20
21   lgdt(c->gdt, sizeof(c->gdt));
22   loadgs(SEG_KCPU << 3);
23
24   // Initialize cpu-local storage.
25   cpu = c;
26   proc = 0;
27 }
```

xv6のアドレス空間のレイアウトは、2GB以上の物理メモリを扱えないという欠点がある。
それは修正可能ではあるが、一番いいのは64ビットアドレスのマシンに切り替える事だろう。

xv6は、240MBと仮定した分の代わりに実際のRAMの設定を決定しなければならない。
x86では、よく知られたアルゴリズムは少なくとも3つある。
1つ目は、事前に用意した値を書きこんで、メモリのように振舞うように見える物理アドレス空間上の領域を走査する方法である。
2つ目は、PCの不揮発なRAMにある知られた16ビットの位置の外にあるメモリのキロバイト数を読み取ることである。
3つ目は、BIOSメモリの中から、マルチプロセッサテーブルの一部のように、残っているメモリレイアウトテーブルを参照することである。
メモリレイアウトテーブルを読み取ることは複雑である。

大昔は、メモリの割り当ての話題で持ちきりだった。
非常に限定されたメモリの使い方を効率化することや、不明な将来の要求に備えることは、基本的な問題だった。
Knuthを見ろ。
今日では、人々は領域の効率化より速度について気にしている。
付け足すと、より複雑なカーネルはほぼ間違い無く、（xv6のように）4096バイトちょうどのブロックではなく、小さいブロックで構成される様々なサイズを割り当てるだろう。
実際のカーネルのアロケータは、大きな領域の割り当てと同じくらい小さな領域の割り当ても扱える必要があるだろう。

execはxv6の中で一番複雑なコードである。
それはポインタの変換を伴い（それはsys_execでも同じである）、エラーになる場合がたくさんあり、実行中のプロセスを他で置き換えなければならない。
実際のOSでは、execの実装はそれ以上に複雑だ。
それらは、シェルスクリプトや（次のexerciseを参照）、もっと複雑なELFバイナリや、複合的なバイナリフォーマットを扱う。

感想

実際のOSにおけるメモリ割り当ての概要ですね。

x86で実際の物理メモリ量を検出する有名な方法が3つあるようです。
詳細は不明ですが、1つ目は全走査する方法、2つ目はメモリチップの情報を読取る方法（多分）、3つ目はBIOSから読み取る方法みたいです。
でBIOSから読み取るのは多分いろんな場合があって難しいんでしょうね。

クヌースを見ろの所はよく分かりません。
クヌースは凄い人なので、彼の考え出したアルゴリズムが今日もメモリ割り当てアルゴリズムとして生かされてるとか、当時メモリ割り当てに関する議論を盛んに行なったりとか普通にありそうですが、具体的な例は知りません。
知ったらへぇ〜ボタンを20連打したくなるようなレベルの事だろうなという気がして、ちょっと調べてみたんですが、これか！というのは見つけれませんでした。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/2/24 金曜日 [http://peta.okechan.net/blog/archives/1310] |