Linux/Alpha 活用講座

清水尚彦　nshimizu@keyaki.cc.u-tokai.ac.jp

Alphaアーキテクチャに最適化したLinuxカーネルパッチの作成(2)

悪態をつきながら使っている「Telios」(シャープのWindows CEマシン)ですが、相変わらず電池切れとともにフルリセットがかかり、そのたびに時間設定からやり直すことになります。しかし、さすがにフラッシュメモリを常に使うようになって、編集中の文章が消えても困らないように自衛しました。あんまりフルリセットが頻繁なので、時間設定はする気にもなりません。この機械、バックアップもしないくせに、バックアップバッテリの警告を出すのですが、どうせ警告を出すならとバックアップバッテリは外してしまいました。何の役にも立たないこの警告を抑止する方法があるといいのですけどね。

カーネルパッチの詳細

さて、9月号で約束した「ページ粒度ヒントのサポート」(Page Granularity Hint support)をカーネルに組み込むパッチを、カーネルのメモリ管理部分の動きとともに解説しましょう。前回書いたように、このパッチは機能と性能の確認のためのものなので、特殊な条件の下でしか有効とならないようにしています。

●カーネルコンフィギュレーション

まずは、カーネルコンフィギュレーションのメニューから変更しましょう。カーネルのコンフィギュレーションにおいて、「実験的なコードを表示させる」と指定しないかぎり、コンフィギュレーションのメニューに出てこないようにします。

コンフィギュレーションのメニューに出てくる項目は、カーネルソースの中のarch/alpha/config.inというファイルに記述されています。実験的なコードを使う場合には「CONFIG_EXPERIMENTAL」のフラグが立つので、今回のパッチは、このフラグが立っている場合にだけ有効にします。

そこで、config.inの「generic」のメニューに、リスト1の1行を加えます。これによって、genericのメニューの中に「PAGE Granularity Hint support」という項目が出てくるようになります(この形式は、2.3.x以降のカーネルコンフィギュレーションで使われる形式なので、2.2.xでは使えません)。

「dep_bool」は、他のフラグに依存してメニューに表示させることを表しています。「bool」となっているので、有効、無効の二者択一のメニューになります。このパッチは、ドライバと違ってカーネル中心部分に変更を加えるため、モジュールとしては実装できないのでこのようにしています。

この機能を有効にすると、コンパイルオプションに「CONFIG_PAGE_GH」という定義が追加されるので、ソースファイルで

```
#ifdef CONFIG_PAGE_GH
......
#endif
```

のように、この機能が必要とするコードを他のコードと条件分けしてコンパイルできるようになります。

リスト1　config.inの「generic」メニューに追加する「CONFIG_EXPERIMENTAL」のフラグ

```
dep_bool 'PAGE Granularity Hint support' CONFIG_PAGE_GH $CONFIG_EXPERIMENTAL
```

●ソースコード解説

本誌2000年9月号で、Linuxのメモリ管理については概略を説明しましたが、今回のパッチで使う部分を少しソースリストの中で追いかけてカーネルの動きを調べてみましょう。

まず、前回説明したように、ページ粒度ヒントの設定は、brkシステムコールでメモリが要求されたときに限定してサポートするとしました。brkシステムコールが呼ばれると、まずはsys_brkという関数が呼び出されます。これは、mm/mmap.cの中にあります。冒頭部分は**リスト**2のようになっています。

メモリを扱うためにcurrent変数の中のmm構造体を参照しているのが分かりますね。

brkは、メモリを確保するだけではなく、それを開放したりもするので、この後の部分で場合分けして処理を行っていますが、我々のパッチに関係する部分は**リスト**3の部分になります。

これを見ると、単にdo_brkという別の関数を呼び出していることが分かります。というわけで、次にこのdo_brkを追いかけます。冒頭部分は**リスト**4のようになっています。

この後にいくつかのエラーチェックが続いて、本体部分は**リスト**5です。

まず仮想メモリが拡張されるので、新しく割り当てられる仮想メモリに対応するvm_area_struct構造体の場所を割り当てます(**リスト**6)。

プログラムから新規に割り当てたメモリに対する各種パラメータを設定します。ここで、新しい仮想メモリは「addr」から始まって、長さ「len」を加えたアドレスが終了アドレスになります。また、プログラムの固有領域として適切なフラグを設定しておきます。ここで、PROT_READからPROT_EXECまでは、後でハードウェアのページテーブルに直接設定されるフラグ(vma->vm_page_prot)に対応します。

ここで覚えておいてもらいたいのは、「vma->vm_flags」には「mm->def_flags」が「ORされている」ということです。後でこの情報を利用して、ユーザープログラムとOSの間の連携をとることにします。

構造体のメンバの設定が終わったら、現プロセスの仮想空間にマージします(**リスト**7)。

この処理を行う前後では、スワップデーモンが勝手に仮想空間を変更したりしないように、ロックをかけています。また、そのときに、前後の空間の状態によっては構造体のマージを行い、なるべく大きな領域で管理できるようにし、同時に、仮想空間の大きさを新規に割り当てた分だけ増やします。この部分にある「PAGE_SHIFT」は、仮想空間のページの長さについて、底が2の対数をとった値になっています。Alphaでは8KbytesのページPAGE_SHIFTは「13」になります。

mlock のシステムコールでページ常駐が指示されているときに VM_LOCKEDのフラグが立ちます。このときにロックされた領域長に、今回コールされた長さを加えます(こちらも例によってページ長が単位です)。その後、実ページを割り当てるためにmake_pages_presentの関数をコールします(**リスト**8)。このフラグ変数は、vm_area_struct構造体をマージする前にフラグの値をセーブしておいたものなのですが、先ほどの構造体メンバの値をセットした部分でmm->def_flags以外は固定値がORされていたことにお気づきでしょうか？　つまり、VM_LOCKEDのチェックは、mm->def_flagsにこのビットが立っているかどうかのチェックだということになります。

リスト2　sys_brk関数定義の冒頭部分(mm/mmap.c)

```
asmlinkage unsigned long sys_brk(unsigned long brk) {
        unsigned long rlim, retval;
        unsigned long newbrk, oldbrk;
        struct mm_struct *mm = current->mm;
```

リスト3

```
/* Ok, looks good - let it rip. */
if (do_brk(oldbrk, newbrk- oldbrk) != oldbrk)  goto out;
```

リスト4　do_brk関数定義の冒頭

```
unsigned long do_brk(unsigned long addr, unsigned long len) {
        struct mm_struct * mm = current->mm;
        struct vm_area_struct * vma;
        unsigned long flags, retval;
```

リスト5　do_brk関数のメイン処理

```
/*
* create a vma struct for an anonymous mapping
*/
vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
if (!vma) return -ENOMEM;
```

リスト6　vm_area_struct構造体の割り当て処理

```
vma->vm_mm = mm;
vma->vm_start = addr;
vma->vm_end = addr + len;
vma->vm_flags = vm_flags(PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_FIXED|MAP_PRIVATE) | mm->def_flags;
vma->vm_flags |= VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
vma->vm_page_prot = protection_map[vma->vm_flags & 0x0f];
vma->vm_ops = NULL;
vma->vm_pgoff = 0;
vma->vm_file = NULL;
vma->vm_private_data = NULL;
```

リスト7　vmaを仮想空間にマージ

```
/*
* merge_segments may merge our vma, so we can't refer to it
* after the call. Save the values we need now ...
*/
flags = vma->vm_flags;
addr = vma->vm_start;

vmlist_modify_lock(mm);
insert_vm_struct(mm, vma);
merge_segments(mm, vma->vm_start, vma->vm_end);
vmlist_modify_unlock(mm);

mm->total_vm += len >> PAGE_SHIFT;
```

リスト8

```
 if (flags & VM_LOCKED) {
  mm->locked_vm += len >> PAGE_SHIFT;
  make_pages_present(addr, addr + len);
  }
  return addr;
```

最後に、brkのアドレスを返して関数を終了します。これから分かるように、実ページはページ常駐を指示したときのほかは割り当てられません。割り当てられなければどうなるかというと、この領域にアクセスするとアクセス例外が発生することになり、アクセス例外の割り込み処理ルーチンの中で実ページの割り当てを行うことになります。さらにいうと、実はページ常駐を指示したとしても本当にページを割り当てるのは限定された条件の下でしかなく、それ以外はやはりページの割り当ては行われません。

今回のパッチでは、ページ常駐をしたときに動作するようにしたので、実際の処理はmake_pages_presentの関数の中で行われます。次にこの関数を追いかけてみましょう。

make_pages_presentはカーネルのmm/memory.cにあります。この関数は大変小さなものですから全文を載せましょう（**リスト9**）。

まず、与えられたアドレスに対応するvm_area_struct構造体を検索してから、1ページずつページ割り当ての関数handle_mm_faultを呼び出します。この関数は、アクセス例外

リスト9　make_pages_present(mm/memory.c)

```
int make_pages_present(unsigned long addr, unsigned long end)
{
  int write;
  struct mm_struct *mm = current->mm;
  struct vm_area_struct * vma;

  vma = find_vma(mm, addr);
  write = (vma->vm_flags & VM_WRITE) != 0;
  if (addr >= end)  BUG();
  do {
    if (handle_mm_fault(mm, vma, addr, write) < 0)  return -1;
    addr += PAGE_SIZE;
    } while (addr < end);
  return 0;
}
```

の割り込みルーチンでも呼び出されます。このままだとページ割り当ては1ページごとになってしまいますね。この関数を直接変更すると割り込みルーチンと共用できないので、別の関数make_new_pages_presentを作ることにします。それに伴って、do_brkでページ常駐のときに呼び出す関数をこの新しい関数に変更します。

Alphaのページ粒度は4種類あるので、アドレスの範囲内でなるべく大きな粒度を使うように工夫します。各粒度ごとに必要な情報をAlpha固有のヘッダ情報の中に定義します。メモリ関連のヘッダファイルはいくつかありますが、ページテーブルの操作の定義をしているinclude/asm-alpha/pgtable.hにリスト10のような行を追加します。

ページテーブルの中の粒度ヒント情報の位置を示すマスク情報と、粒度のバリエーションの数とを定数として定義します。そして、各粒度のページ数の2を底とする対数を取った値(オーダー)を配列としてpage_gh_orderに格納し、粒度ごとページテーブルへ設定する値を、同じく配列として用意します。配列の中身は、それをヘッダファイルで格納するとインクルードした全部のファイルで格納場所を用意することになって無駄だらけになるので、メモリ管理の初期化ルーチンの中で行います。これはarch/alpha/mm/init.cです(リスト11)。

さて、いよいよページ粒度に対応したページの割り当てルーチンのメイン部分です。この関数の元となる関数make_pages_presentから呼び出していたhandle_mm_faultは、1ページだけの割り当てルーチンだったので、オーダー単位での割り当てルーチン「handle_ mm_falut_pages」という関数を新たに作ることにします(リスト12)。

この関数では要求された粒度(step)で定義された大きさの実ページをalloc_pages関数で割り当てた後、ページのプロテクション情報をまとめてセットします。実験的なパッチなので

リスト10　include/asm- alpha/pgtable.hに追加する定義

```
#if defined(CONFIG_PAGE_GH)
#define PAGE_GH_MASK 0x0060
#define PAGE_GH_NR 4
extern int page_gh_order[];
extern pgprot_t page_gh_prot[];
#endif
```

リスト11　ページ粒度情報の配列定義(arch/alpha/mm/init.c)

```
#ifdef CONFIG_PAGE_GH
 int page_gh_order[] = {0,3,6,9};
 pgprot_t page_gh_prot[] =
{0x0000,0x0020,0x0040,0x0060};
#endif
```

プロテクション情報はvm_area_structを参照せずに直接作っています。プロテクション情報をセットするために利用するremap_page_range関数は、予約されたページのプロテクション情報しかセットしないようになっているので、この関数も少し変更し、ページ粒度のビットが立っていればページテーブルのエントリにプロテクション情報を設定するようにしています(リスト13)。

また、本来なら、セキュリティ上の観点から、割り当てたページは「0クリア」しておく必要があるのですが、実験的なパッチということでサボっています。

次に、mm構造体の中の実ページサイズを割り当て分だけ増加させておきます。alloc_pagesの中では、オーダー単位で割り当てをした場合にはそのオーダーに対応したページ管理情報に使用ビットが立つのですが、今回のパッチでは、ページの開放は1ページ単位で行われるので、実はうまく開放できません。一般ユーザーが普通に使える正式版のパッチでは、ページ

単位の使用ビットを立てるようにしていますが、今回のパッチのままでは、実はメモリリークになっていて、ベンチマークを動かすにはいいのですが、実使用は難しくなります。ですが、粒度サポートの原理を示すためには、細かなところを割愛したほうが分かりやすくなるので、あえてそのまま掲載します。

それでは以下に、do_brkから呼び出す新しいページ割り当て関数「make_new_pages_present」を示します(**リスト**14)。この関数は、元となったmake_pages_present関数と異なり、実ページを割り当てるためにhandle_mm_fault_pagesを呼び出すようにしています。

ページ粒度のヒント情報は、仮想アドレス、実アドレスともに、粒度の境界に合わせて割り当てないといけません。しかし、一般のプログラムが呼び出すときに仮想アドレスの連続性が保たれるようvm_area_struct構造体を調整しているので、大きな粒度の境界に合わせてこの関数が呼ばれることは、あまりありません。そのため、半端な境界から呼ばれたときに、どれだけ大きな粒度の割り付けを増やすかが、課題となります。そこで、このプログラムでは、ページ粒度の小さいほうからチェックして、1つ上の粒度の割り当てができないときには、現粒度の割り当てを行うようにし、段階的に粒度を上げていくようにします。アドレス境界が最大の粒度に一致しても割り当て範囲の大きさが粒度に合っていなければ、段階的に粒度を下げて一致するところまで検索します。粒度の配列には最小1ページに相当するエントリがあるため、この検索は必ず成功します。

このように、前後の境界処理をしているため、プログラムからはあまり境界条件を考えずにメモリの要求ができるようにしています。もちろん、なるべく大きな単位で呼び出したほうが無駄が少ないため、一度に多くのメモリを割り当てて、必要ならそのエリアをプログラムで分割して利用するような方法を取ると便利でしょう。

A カーネルパッチの効果は？

さて、カーネルの動きやパッチについて解説してきましたが、それよりも性能に興味がある方も多いでしょうから、簡単なベンチマークの結果を示しましょう(**表**1)。私は1人で占有しているEB164でしか動作させていませんが、新しいEV6の系列のプロセッサならば、もっと性能は上がるはずです。みなさんも、ぜひ試してみてください。

リスト12

```
int handle_mm_fault_pages(struct mm_struct *mm, struct vm_area_struct * vma,
  unsigned long address, int write_access, int step) {
  struct page *page;

 if(step == 0)  return (handle_mm_fault(mm, vma, address, write_access) );
  page = alloc_pages(GFP_HIGHUSER, page_gh_order[step]);
  if (page == NULL)
     return -1;
  remap_page_range( address, (page - mem_map) << PAGE_SHIFT,
                    PAGE_SIZE << page_gh_order[step],
                    _PAGE_NORMAL(__DIRTY_BITS | pgprot_val(page_gh_prot[step])));
  flush_tlb_page(vma, address);
  mm->rss += 1 << page_gh_order[step];
 /* you must clear the pages for the release level kernel */
  return 0;
 }
```

リスト13

```
@@ -716,8 +724,10 @@
  pte_clear(pte);
  mapnr = MAP_NR(__va( phys_addr));
- if (mapnr >= max_mapnr || PageReserved(mem_map+mapnr))
+ if (mapnr >= max_mapnr || PageReserved(mem_map+mapnr) || +(pgprot_val(prot) & PAGE_GH_MASK)) {
  set_pte(pte, mk_pte_phys(phys_addr, prot));
+ }  forget_pte(oldpage);
address += PAGE_SIZE;  phys_addr += PAGE_SIZE;
```

リスト14

```
int make_new_pages_present(unsigned long addr, unsigned long end) {
  int write;
  struct mm_struct *mm = current->mm;
  struct vm_area_struct * vma;

 vma = find_vma(mm, addr);
  write = (vma->vm_flags & VM_WRITE) != 0;
  if (addr >= end)  BUG();
  do {
  int i;
  unsigned long rem;
  for (i = 0; i< PAGE_GH_NR - 1; i++) {
     rem = (~addr + 1) & ((PAGE_SIZE << page_gh_order[i+1]) - 1);
     while (rem
         &&  (addr & ((PAGE_SIZE << page_gh_order[i]) - 1)) == 0UL
         &&  ((end - addr ) >= (PAGE_SIZE << page_gh_order[i]))) {
         if (handle_mm_fault_pages(mm, vma, addr, write, i) < 0)
            return -1;
       addr += PAGE_SIZE << page_gh_order[i];
       rem -= PAGE_SIZE << page_gh_order[i];
       };
  }
  for (i = PAGE_GH_NR - 1; i >= 0; i--) {
     while (  (addr & ((PAGE_SIZE << page_gh_order[i]) - 1)) == 0UL &&
           ((end - addr ) >= (PAGE_SIZE << page_gh_order[i]))) {
       if (handle_mm_fault_pages(mm, vma, addr, write, i) < 0)
         return -1;
       addr += PAGE_SIZE << page_gh_order[i];
     };
  }
 } while (addr < end);
 return 0;
}
```

表1　粒度サポートカーネルの効果

ベンチマーク	粒度サポートカーネル	通常のカーネル	性能向上率
配列転置	51Mbytes/秒	36Mbytes/秒	41%
行列乗算	254MFLOPS	208MFLOPS	22%
FFT	0.30秒	0.59秒	97%

このように、プログラムの工夫がされていなくても簡単に性能が向上することが分かります。このパッチが特に有効なのは、メモリ空間を広く使うHPCアプリケーションや、ポインタを追いかけていくようなLISP系のアプリケーションです。特殊な条件なしに使える正式なパッチをカーネルメーリングリストに投稿しているので、うまく次期バージョンに取り入れられると、多くのユーザーの便利がいいのですけどね。どうなることでしょうか。

たった10Kbytesのパッチでこれだけの性能向上が得られるのは面白いですね。同じ考え方で、Athlonなどのラージページもサポートできると思いますが、x86を使う機会はほとんどないので深く検討はしていません。

前回と今回の2回に分けて、実験的なカーネルパッチとその動作について解説してきました。正式版のパッチも完成したので、次回は引き続き、正式版のカーネルパッチについて設計思想と新しいパッチの説明をしたいと思います。