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(3)

ループバックアクセスを高速化するパッチ

私は、LSI設計のための演習授業を春と秋に2コマずつ持っていますが、その中で使っているソフトウェアのために、Linuxを大学の計算センターにインストールすることにしてもらっています。今年は、センターのシステムのリプレースでPCも一緒にリプレースされるので、120台のマシンへLinuxをインストールしなくてはなりません。これを2、3人で1日で仕上げる必要があるので、システムの計画には結構気を使います。

前回は、UMSDOSのファイルシステムの上に構築したシステムイメージをtarでアーカイブして、一旦センターのサーバに入れたあと、各マシンでダウンロードして、DOS版のtarで各マシンごとに展開していました。今回は、UMSDOSよりましな方法をと思って、ループバックファイルシステムを使って、利用環境を立ち上げようとしています。initrdからループバックを呼び出して動作するようにし、ユーザーホームディレクトリは、RAMディスクに確保しておくことで、立ち上げのたびに同じ環境を提供できます。

プロトタイプ設計用のマシンも貸し出され、早速、環境を設定していたのですが、ループバックの性能がどうも思わしくありません。ベンチマークを取ってみると、「ext2 on FAT」は、ext2に比べて、システム時間で10倍近くかかっていることが分かり、チューニング手段はないかと探していました。

fj.os.linuxで相談してみたところ、メディアラボのMLD4では独自パッチで高速化していると教わりました。そこで、MLD4のメーリングリストで問い合わせたところ、快くパッチを公開して頂きました(予算やさまざまな事情で有償の商用のパッケージを使うのは難しいので、MLD4を購入することはできなかったのです)。

これは、RAIDなどを実現するmdドライバと同様に、ブロックデバイスのドライバの中で、直接ブロックマップを行うもので、なかなかの優れモノです。私と同じような悩みを抱えている人は(そんな奴いないって?)お試しください。システム時間のオーバヘッドは10%程度と、実使用に障りがない程度に改善されました。

ページ粒度パッチの改良

先月号でお約束した通り、今月は、普通のプログラムも高速化できるように「ページ粒度のパッチ」を修正していきたいと思います。

といっても、執筆時点ではまだ完全ではなく、bibtexや一部のXサーバがトラブルを起こしています。しかし多くの高性能計算のプログラムは問題なく動いていますので参考になると思います。ELFローダのbss領域をサポートしなければ動くのですが、これを止めると、FORTRANが確保する配列領域が対象外となってしまうので、今、問題点を追求しているところです。自分の作ったところを自分でチェックしても、思い込みがあるので、なかなかデバッグは進まないのですが、問題点が解決したら筆者のホームページ(記事末RESOURCE[1]を参照)で情報を公開しますので、利用する人はときどきチェックしてみてください。

なお余談になりますが、どんなものでも、デバッグの効率は可観測性に大きく左右されます。どれだけ有効な情報を容易に取得できるようにするかが問題になるのです。ユーザープログラムならば、gdbでトレースという最終手段が取れますが、カーネルのメモリ管理の部分では、途中で止めたりダン

プしたりすることは難しく、適宜挿入したprintkで必要な情報を画面に出力する方法が中心になります。このときも、ただやみくもに出力しても仕方がなく、プログラムで条件を判定して必要なところだけ出力するなどの工夫が必要です。再現性の高いバグの場合には、これだけでかなり潰すことができます。

先月号のパッチは2.4.0-test4のカーネルを対象にしていましたが、その後、ビジュアルテクノロジー社からお借りしている「DP264」でもベンチマークをするため、(SMPでは動かなかった2.4.0-test4でなく)2.2.16用のパッチも急拠作成しました。2.4.0-test4のSMPの問題はすでに解決されているので、こちらでもいいのですが、動作の確認のためには、実績のあるパッチ前のカーネルとの比較が必要になるので、今月号では、2.2.16用のパッチを対象に話を進めたいと思います。2.4.0-test?用のパッチも順次メンテナンスしていきますので、最新版を使いたい方は、私のホームページ([1])をごらんください。

先月号で示した方式と今回の方式は、プロセスの実行面から見ると、大きく変わっています。一番の変更点は、前回は特殊な条件の中でのbrkシステムコールが来たことをきっかけにして粒度の大きな領域をあらかじめ割り当てていたのに対して、通常のLinuxのメモリ管理と同様に、メモリ要求があったときには仮想空間だけを拡張しておき、プロセスが実際にページを参照したときに初めてページを割り当てるというところです。これによって、すぐに利用されない領域にメモリを割り当てることがなくなり、メモリの利用効率が向上することになります。通常のメモリ割り当ての流れを簡単に図解したものが図1です。

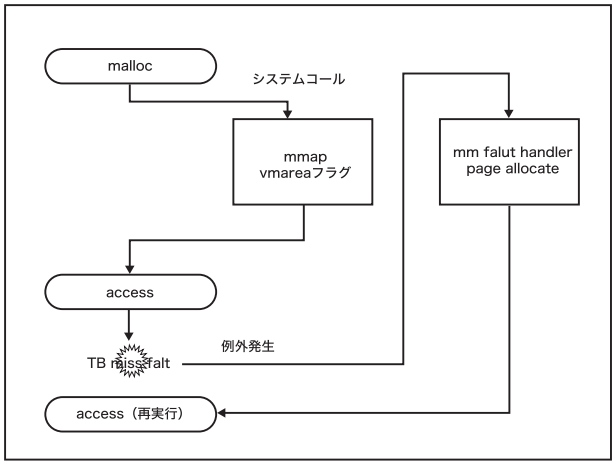

図1

プロセスがmalloc関数でメモリを要求すると、システムコールのmmapが呼び出されます。このシステムコールは、プロセスの仮想メモリの領域を修正しますが、実際のページテーブルの修正はしないので、実ページを割り当てることはしません。そこで、mallocで割り当てられた領域をプロセスがアクセスすると、ハードウェアの変換バッファ(TBもしくはTLB)は例外を発生させることになります。例外のハンドラはカーネルの中にあって、vmareaのフラグを見て、この例外が発生したアドレスがすでにプロセスに割り当て済の領域を示しているのであれば、新たにページを割り当ててページテーブルを調整し、例外から復帰します。プロセスは、例外を起こした命令を再実行することで、割り当てられた領域に、今回は例外を起こすことなくアクセスできます。

プロセスが大きな粒度でアクセスできるような要求を出しているかどうかは、mallocから呼び出されたmmapでは把握できますが、アクセス例外の時点では、例外を起こしたまさにそのページの情報しか得られないため、大きな粒度で割り当てることはできません。

そこで、今回のパッチでは、mmapからページ例外ハンドラに粒度の情報を渡すことで、アクセス例外の処理の中で、大きな粒度のページが割り当てられるようにします。情報を渡すといっても、数多くのプロセスが同時に走っているLinuxの実システムで、プロセスごとの情報を渡す方法が必要になりますから、何らかの方法が要求されます。この情報は、仮想空間の各ページごとに設定する必要があります。というのは、アクセス例外が割り当てた仮想空間のどの部分から発生するかは、プロセスがどのようにメモリを使うかによって変わり、あらかじめ予測することはできないからです。

今回は、ページテーブル自身に情報を記憶する方法を採用しました。プロセッサにとっては、ページテーブルは有効か無効かが大事で、無効になっているテーブルに何が記憶されていてもあまり関係ないからです。ところが、今まで述べてきたように、プロセスが仮想空間を要求するときには、ページテーブルすら用意されていないので、そのままのLinuxにこの方法を使えません。そこで、mmapで新規に仮想空間を割り当てるときにページテーブルまでは合わせて作ってしまい、大きな粒度のページには粒度情報を設定することにします。これによって、アクセス例外が発生したときに、例外を起こしたページを見て大きな粒度の割り当てが可能か否かを判定できるようになります。

ところで、Linuxは、無効なページテーブルエントリにスワップのための情報を記憶するので、これとぶつからないようにしなくてはなりません。Alphaでは、スワップの情報は

ページテーブルの上位32bitに記憶されます。下位のビットはスワップとして使っているときには「0」が入っています。この上位32bitは、通常のページテーブルではページフレーム番号が記憶されることになっていて、下位にはページ保護情報が記憶されます。また、ページの粒度の情報も保護情報の中の一部に設定するようになっています。

ページテーブルといえども、貴重なメモリ資源を使うわけですから(といっても昔のマシンほどメモリも高価ではないのですが)、無駄に割り当てるのは問題です。ページ粒度ヒントを使うときの一番の制約は、ブロックの中のページがすべて同じプロテクション情報を共有しなくてはならないということです。ところが、要求された仮想空間のプロテクション情報は、しばしば変更される場合があって、そのときには、そのブロックの中のページ粒度ヒントをすべて消して回る必要が出てきます。

これは馬鹿にならないオーバヘッドになるので、実際の割り当ては、ユーザーが要求している仮想空間の中でも、できるだけページテーブルの情報がすぐには変更されにくいと思われる空間に限りたいものです。そこで、プロテクション情報が一番緩いmallocタイプのメモリ割り当てに限定して粒度のヒントを設定するようにします。C言語のプログラムでは、これで十分な高速化が図れます(FORTRANの配列や、C言語でも大域変数に取った領域は、プロセスのbss領域に割り当てられます。これをサポートするには、fs/binfmt_elf.cの対応する部分を変更する必要があります)。

ページテーブルに対するページ粒度の設定は、仮想空間と実アドレス空間の双方で、粒度に合わせた境界アドレスにアラインされている必要があります。これは、前回のパッチと同じように、仮想アドレスの境界を調べながら、順次大きな粒度の割り当てをしていくことで対応できます。

今回のパッチの一番トリッキーな部分は、粒度の設定をされたページテーブルのプロテクション情報が変更されたときに、同ブロック内のすべての粒度情報をクリアする部分です。粒度設定がされているかどうかは、古いページテーブルを読み出すことで把握できるのですが、Linuxカーネル中に発生するすべてのページテーブル書き込みを対象に、この検査ルーチンを入れなくてはなりません。カーネルリリースの将来に対しても保証できるような方法で検査ルーチンを入れる必要があるのですが、幸いなことにLinuxでは、ページテーブル設定に特殊な処理が必要なプロセッサ向けに、ページテーブル設定をマクロ「set_pte」として定義しています。また、ページテーブルをクリアするインライン関数「pte_clear」も、ページテーブルを変更する関数として上げることができます。そこで、このマクロとインライン関数を利用して、検査ルーチンを入れることにします。

まず、粒度情報を新たに設定するのは、先ほど書いた粒度情報だけを記憶するページテーブルを作成する場合と、アクセス例外を受けて実際にページを割り当てて、その内容に従ったページテーブルエントリを書き込む場合です。それ以外のページテーブル書き込みは、すべて粒度情報をクリアする要求とみなして構いません。これら2つのルーチンと、そのサポート関数をインライン関数として作成します。粒度情報クリアのインライン関数と合わせて、以下のような関数を用意しました。

・pte_t mk_pte_gh_clean(**リスト1**)

このルーチンは、粒度情報をクリアしたページテーブルエントリを作成するための補助関数です。

・clear_pte_gh(**リスト2**)

ページテーブルの粒度境界からブロック内のすべてのエン

リスト1

```
extern inline pte_t mk_pte_gh_clean (pte_t pte) {
  pte_val(pte) &= ~PAGE_GH_MASK;
  return pte;
}
```

リスト2

```
extern inline void clear_pte_gh (pte_t *pteptr, int order) {
  int i;
  pte_t *addr = (pte_t *)((unsigned long) pteptr & ~((1UL<<(order +
SIZEOF_PTR_LOG2)) - 1));
  for ( i=0; i < 1<<order; i++)
    pte_val (*(addr+i)) &=  ~PAGE_GH_MASK;
}
```

トリの粒度情報をクリアします。

・set_pte_raw(リスト3)
旧ページテーブルエントリに粒度情報が設定されていて、そのページが有効ならば、ブロック内の粒度情報をクリアしたのち、新たなページテーブルエントリを書き込みます。ページが無効であったり粒度情報がもともと設定されていないエントリには、そのまま新しい情報を書き込みます。

・set_pte(リスト4)
呼び出された引数の粒度情報をクリアしてページテーブルの設定関数を呼び出します。

・pte_clear(リスト5)
ページテーブルエントリをクリアするために、set_pteを呼び出します。この関数は、元々は、直接ページテーブルエントリをクリアしていたのですが、粒度情報をクリアする必要がある場合があることから、このようにしています。

do_anonymous_page

さて、プロセスがアクセス例外を起こしたときにページを割り当てる役割を担っているのが「do_anonymous_page」という関数です(リスト6)。これは、mm/memory.cにあります。この関数は、本来は例外を起こしたアドレスに1ページの実メモリを割り当てているのですが、割り当てようとしたページテーブルに、前出の方法で粒度情報が設定されていた場合には、その粒度に見合ったサイズのページを割り当てるように変更しましょう。

リスト3

```
extern inline void set_pte_raw (pte_t *pteptr, pte_t pteval) {
  if ( pte_present(*pteptr) && ( pte_val(*pteptr) & PAGE_GH_MASK ))
    clear_pte_gh(pteptr, page_gh_order[pte_to_gh_index(*pteptr)]);
  *pteptr = pteval;
}
```

リスト4

```
extern inline void set_pte(pte_t *pteptr, pte_t pteval) {
       set_pte_raw(pteptr, mk_pte_gh_clean(pteval));
}
```

リスト5

```
extern inline void pte_clear (pte_t *ptep) {
  pte_t pte;
  pte_val(pte)=0;
  set_pte(ptep, pte);
}
```

リスト7

```
void __break_area (int type, struct page *page, unsigned long order) {
  unsigned long size = 1 << order;
  unsigned long flags;
  int i,j;
  unsigned int index = (page - mem_map);
  struct free_area_struct * area = free_area[type];
  spin_lock_irqsave(&page_alloc_lock, flags);
  for ( i = 0; i < size; i++ ) {
    atomic_set(&(page+i)->count, 1);
  }
  spin_unlock_irqrestore(&page_alloc_lock, flags);
  return;
}
```

まず、例外を起こしたページの粒度情報をチェックします。そして、この粒度情報に合わせて実メモリを要求します。

ここで実メモリが取得できないときに大きな粒度を要求している場合、一旦粒度情報をクリアしてから、再度do_anonymous_pageを呼び出します。これによって、1ページだけの要求に変わるため最後の1ページまで割り当てることができます。大きな粒度で要求したページは、その粒度に応じた利用ビットがセットされるのですが、今回の方式では、ページの開放は粒度と関わりなく、1ページ単位で行われることになるので、実メモリの管理ユニットに1ページごとの利用になっているように見せかけなければいけません。そのための関数を「break_area」として、別途用意しました。その後、1ページごとにページテーブルを作成し、そのページをクリアしておきます。

break_areaは、wrap関数で、その本体はリスト7のようになっています。粒度で示されたページ数分だけ利用カウントに「1」を設定します。

リスト6 do_anonymous_page関数

```
static int do_anonymous_page (struct task_struct * tsk,
                        struct vm_area_struct * vma,
                        pte_t *page_table,
                        int write_access,
                        unsigned long addr) {
  int i;
  pte_t entry = pte_wrprotect(mk_pte(ZERO_PAGE(addr), vma->vm_page_prot));
  pte_t oldpte = *page_table;
  int order = page_gh_order[pte_to_gh_index(oldpte)];

/*
 * We allocate a block of pages if the oldpte has the GH bit. This
 * routine is called with empty oldpte but GH bits.
 */
  if (write_access) {
    pte_t *wktable = (pte_t *)((unsigned long)page_table & ~((1UL << (order + SIZEOF_PTR_LOG2)) -1));
    unsigned long page = __get_free_pages(GFP_USER, order);
    if (!page) {
      if (order) {
     clear_pte_gh(page_table, order);
     return do_anonymous_page(tsk, vma, page_table, write_access, addr);
      } else
     return -1;
    }
    if (order) break_area(GFP_USER, page, order);
    for (i=0; i < 1 << order; i++) {
      entry = pte_mkwrite (pte_mkdirty
                     (mk_pte (page+(i<<PAGE_SHIFT),
                             __pgprot(pgprot_val(vma->vm_page_prot) |
                                    pgprot_val(page_gh_prot[pte_to_gh_index(oldpte)])))));
      clear_page(page+(i<<PAGE_SHIFT));
      vma->vm_mm->rss++;
      tsk->min_flt++;
      set_pte_raw(wktable+i, entry);
      flush_page_to_ram(page+i);
    }
  } else
    put_page(page_table, entry);
  return 1;
}
```

まとめ

最初に書いたように、まだこのパッチは完全ではないので、今回の解説は、ページ粒度ヒントサポートの考え方を中心にまとめました。この成果もいろいろと条件によって変わってくるので、最終的な値ではありませんが、とりあえず表1のようなデータが得られています。この値は、ビジュアルテクノロジー社の「DP264(667MHz)」で取得したものです。

このように、実行するプログラムによって、大きく性能が変わりますが、FFTや転置のように、広い範囲のメモリをアクセスするもので良い結果が出ていることが分かります。連続アクセスの代表のような「姫野ベンチ」にしても、7%程度、性能が向上していることから、粒度サポートは有意に性能が向上しているといえるでしょう。

問題点を早急に対策して、なるべく早く正式版としてリリースしたいと思っています。筆者のホームページに進捗を載せていきますので、興味のある方は参考にしてください。この方式は、おそらく、Athlonのように複数の粒度のTLBを持つ他のプロセッサにも有効だと思っています。これらのプロセッサを持っている方は、ぜひこれを参考に、粒度サポートパッチの作成を試してみてください。

表1

ベンチマーク	粒度サポートあり	粒度サポートなし	向上比
行列乗算(MFLOPS)	801.4	747.2	1.07
配列転置(MB/S)	282.69	81.3	3.48
FFT(秒)	1.65	2.09	1.267
姫野ベンチ(MFLOPS)	190.5	177.5	1.07

RESOURCE

[1] 筆者のホームページ
http://shimizu-lab.et.u-tokai.ac.jp/~nshimizu/

COLUMN

LinuxによるLSI設計教育

私が勤務している東海大学通信工学科では、次のような授業をLinuxを用いて行っています。

集積回路設計製作I

CMOS回路とLSI設計の基礎演習を行う授業で、具体的には以下のような内容を扱います。

①CMOS回路を回路シミュレータSPICEによって検証する
②基本ゲートレイアウトをレイアウトエディタMagicで作成する
③Magicで作成したレイアウトからネットリスト(回路情報)を抽出しSPICEで検証
④基本ゲートレイアウトを組み合わせた大規模LSIをMagicで作成する
⑤作成したレイアウトの評価と速度改良

集積回路設計製作II

こちらは、RISC型マイクロプロセッサの設計を行う授業で、具体的には以下のような内容を扱います。

①ハードウェア記述言語による組み合わせ回路の設計と論理シミュレータでの検証
②ハードウェア記述言語による順序回路の設計と論理シミュレータでの検証
③ハードウェア記述言語によるマイクロプロセッサの設計と論理シミュレータでの検証
④設計したプロセッサのFPGA(Field Programmable Gate Array)での実現
⑤作成したプロセッサの評価と速度改良

こうした授業では、テキストベースの処理が多いので、エディタとしてviを推奨しています(私の趣味です)。しかし、UNIX系のOSに慣れていない学生が大半なので、ファイルの処理やエディタの扱いに、若干の戸惑いがあるようです。しかし、多くの学生がなんとか最後の課題までこなしています(今回のシステムは、少しでも学生の負担を減らすためにWindowsと操作性が近いエディタと、日本語メニューが充実しているKondara MNU/LinuxのFTP版をベースにしたので、少しは楽になるかもしれないと期待しています)。マイクロプロセッサの設計では、苦労して設計したプロセッサのIOピンに接続したトレーニングボード上の発光ダイオードが、自分の用意した命令の実行によって発光することで、物作りの感激を味わっている学生が多くいます。最後のFPGAへのマッピングの部分は、FPGAメーカーのソフトを使わなくてはできないので、Windowsにスイッチしますが、ほとんどの設計フェーズはLinuxで問題なくこなすことができます。

これらの授業の資料の一部は、私の研究室のホームページからご覧になれます。

http://shimizu-lab.et.u-tokai.ac.jp/class99/

興味のある方はぜひ参照してみてください。 (清水尚彦)