第7章 Linuxオペレーティング・システム

見本

本章では，第6章で述べたIA-32アーキテクチャのハードウェア上で，OSがどのように働いているかを説明していきます．説明するための例としてLinux Kernel 2.6.8を用います．説明のためのコードはLinuxに含まれないものもありますが，論理動作の理解に役立てるために挿入してあります．

また，説明をわかりやすくするため，部分的にコードを削除したところもあります．ソース・コードはDebian Sargeのカーネル・ソース・パッケージを利用しています．

7.1　OSが利用する部品

OSが頻繁に利用する基本的なルーチン(部品)を説明します．

● OSによる表の操作――リンク・リスト

▶ 表の操作をどのようにするか

OS内部では，多数の表が特定のタスクに関連しています．これらの表は互いに関連し合うことでタスクの状態を表します．たとえば，それぞれのタスクの基本情報をもつタスク制御表は，複数のタスクの連係を行うための情報をもち，タスクの実行順や条件待ちの状態を制御するのに使います．OSの仕事とは，ほとんど表の処理といってもよいくらいです．

ただし，OSの表の処理では，表の並べ替えや移動を行いません．というのは，実際に表を並べ替えたり移動するとなると，非常に多くのデータ(メモリ)の移動となり，時間の消費がとても大きくなります．そこでOSでは，表の連係の操作を，リンク・リストの形式を用いて行います．

リンク・リストでは，表を移動するのではなく，表の接続を変えることにより，並べ替えや移動と等価的な処理ができます．これだと，処理時間を最小限におさえることができます．

▶ リストとは，リンクとは？

ある一組のデータの集まりで，そのうち一部がポインタ型の変数となっていて，そのポインタが別のデータの集まりとの関係を示しているとき，それらつながったデータ全体をリストといいます．一方，リンクとは，リストとなったデータ項目の参照関係をいいます．こうした方式を使えば，つねに変化していくデータに対して柔軟な表(データ構造)がつくれます．

● 単方向リンク

表間のもっとも基本的な連係は，単方向リンクと呼ばれるものです．**図7.1**にその例を示します．この**図7.1**のように，単方向リンクとは，先頭から後方へと一方向に表を1列につなげる方式です．

単方向リンクを作るためには，基点を示す変数が必要です．**図7.1**では，その変数をtopと名付けています．リンクが空のときは，topには“0”が記入されます．これが，初期化された状態です．

▶ 単方向リンクの挿入

リンクの挿入は，**図7.1**に示した手順で行います．topが“0”のときは，表が何もつながっていないということなので，無条件にtopの次に接続します．

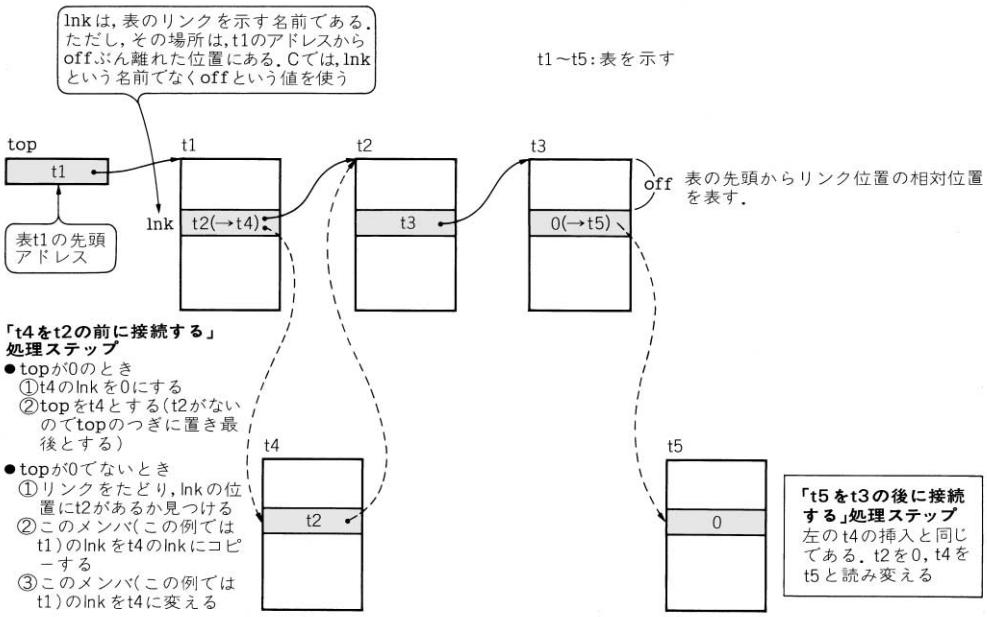


図7.1 単方向リンクにメンバを追加する

リスト7.1　単方向リンクでの挿入ルーチン

```
u_lnk_ins(top,off,loc,ins)      /* 単方向リンクの挿入 */
int  **top;   /* リンクの先頭を保持する変数へのポインタ */
int  off;     /* 表内でのリンクの位置　ポインタ換算 */
int  **loc;   /* 挿入場所 */
int  **ins;   /* 挿入すべき表 */
{
     for(;*top != 0;top = (int **) *top + off) {
          if((int **) *top == loc) {
               *(ins + off) = *top;
               *top = (int *) ins;
               return(0);    /* 登録完了 */
          }
     }
     *top = (int *) ins;     /* リンクの最後に連結 */
     *(ins + off) = 0;
     return(0);              /* 登録完了 */
}
```

(**a**) 挿入ルーチン

```
u_lnk_del(top,off,del)          /* 単方向リンクの削除 */
int  **top;   /* リンクの先頭を保持する変数へのポインタ */
int  off;     /* 表内でのリンクの位置　ポインタ換算 */
int  **del;   /* 削除すべき表 */
{

     for(;*top != 0;top = (int **) *top + off) {
          if((int **) *top == del) {
               *top = *((int **) *top + off);
               return(0);  /* 削除完了 */
          }
     }
     return(1);              /* 削除すべき表がない */
}
```

(**b**) 削除ルーチン

一般にリンクの挿入では，どこに入れるかを指定します．

リンク挿入の関数をC言語で表現します〔**リスト7.1**(**a**)〕．この関数の引き数は3個で間に合いますが，この例ではoffと名付けた引き数を追加して4個とします．

① 先頭のtopを指すデータ(top)
② リンクの相対位置(off)
③ 挿入場所を指すデータ(loc)
④ 自分自身を指すデータ(ins)

このoffと名付けた引き数は，この関数をOS内で汎用的に使いたいため，リンクする位置を名前で決めずに，表の先頭アドレスからのオフセット値で決めることにしています．つまり，表の先頭からこのoff分だけオフセットした位置にリンク変数があることを示します．タスク制御に用いるリンクのオフセットはdefine文で定義されています．

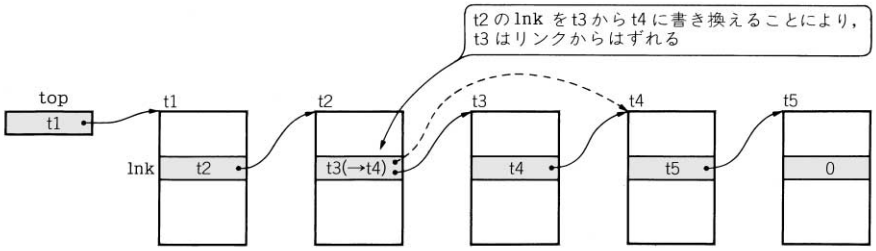


図7.2
単方向リンクの表の除去

「t3をリンクからはずす」処理ステップ
①リンクをたどり lnk に t3 がある表を見つける(この例では t2).
①t3 の lnk の値をこの表(例では t2)の lnk に移す.

単方向リストで挿入を行うためには，まず挿入場所を見つけます．それには，topの位置から，リンクされた経路をたどりながらlocを見つけるまで検索します．もし，locが見つかれば，その前にinsを接続します．locが見つからないときは(リンクの最後は“0”になっている．“0”があればリンク終了と判断する)，リンクの最後に接続を行います．

▶ 単方向リンクでの除去

図7.2は，**図7.1**と逆の操作です．リンク・リストから指定された表を取り去ります．この手順は図に示したように，topから検索し，除去すべき表が見つかったところでリンクの更新を行います．

リスト7.1(**b**)に示した単方向リンク除去関数u_lnk_delの引き数は，

① リンクの先頭(top)
② リンクの相対位置(off)
③ 除去すべき表(del)

です．この関数の戻り値が“0”でないときは，除去すべき表がなかったことを表します．

● 双方向リンク

双方向リンクは，自分の前後のリストに対してリンクをもっています(**図7.3**)．このため，単方向リンクとは異なり，先頭から検索せずにリスト全体を見渡せます．Linuxのリストは，**リスト7.2**(**d**)に示したlist_head構造体を用いて実現しています．これは，前方向のリンクnext(list_headに対するポインタ)と後方向のリンクprevで構成されています．**図7.3**(**a**)のように，リンクはlist_headに対して行われています．このためlist_headがメンバになっている構造体本体のアドレスを得るため，**リスト7.2**(**a**)と**リスト7.2**(**b**)のコードが利用されます．

図7.3(**a**)の例では，構造体t1(構造体tの実体)のメンバであるlがlist_head構造体です．t1のアドレスはlist_entry(ptr,t,l)で得られます(ptrはt1のlを指すポインタ)．

双方向リンクの用い方は，開始点となる構造体が決まっていれば**図7.3**(**a**)のようにループ状にしておくことができます．または，リンクの終端を表すために自分自身へのリンクとすることもできます．たとえばt3のl.nextはt3のl.nextとするなどです．つまり，t3より先がないことを表します．

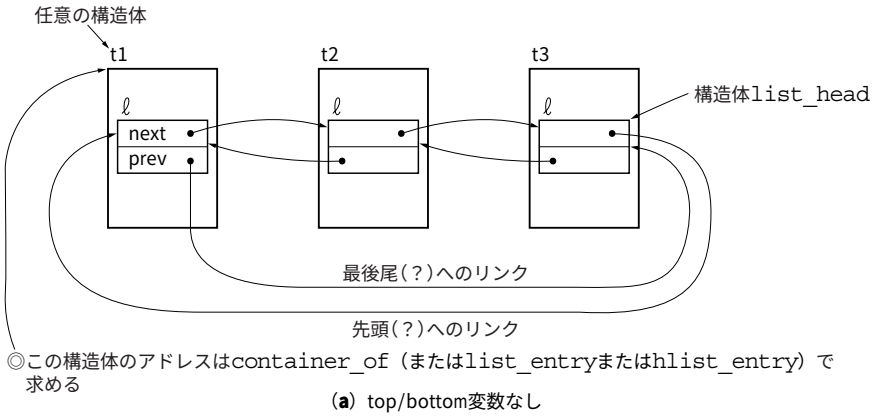
構造体が動的に生成され1個もない状態が存在するときに，top/bottom変数を導入して先頭と終端を決めます〔**図7.3**(**b**)〕．

nullポインタは未定義のポインタを意味しますが，これより強く未定義を主張するため**リスト7.2**(**c**)のようなページ例外を発生するポインタも用意されています．

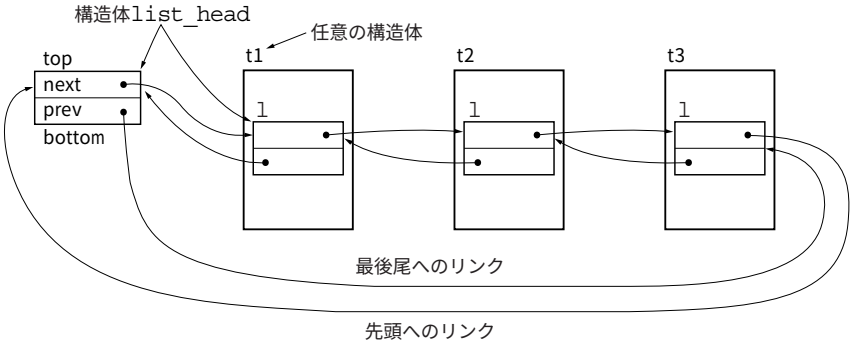
このポインタでlist_headを初期化しておけば，そこにアクセスしたことをOSに知らせることができます．list_headの初期化は，**リスト7.2**(**e**)のINIT_LIST_HEADにより行われます．list_head構造体のメンバは自分自身を指すように初期化されます．

▶ 双方向リンクの挿入

図7.4(**a**)は，リストl2とl3の間にlaを追加する例です．これは，**リスト7.2**(**e**)の__list_addによって行

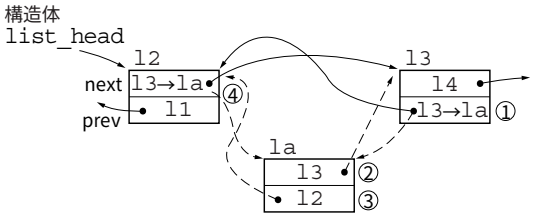


（**a**）top/bottom変数なし



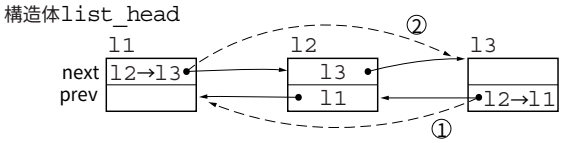
（**b**）top/bottom変数あり

図7.3
双方向リンクのリスト



基本は__list_add（la，l2，l3）で行う．
laがl2とl3の間に接続される

（**a**）双方向リンクの追加



基本は__list_del（l1，l3）で行う．
ℓ_2が除去される

（**b**）双方向リンクの除去

図7.4　双方向リンクの追加・除去

われます(基本)．

図7.4(**a**)に示したように，①～④のステップが実行されます．これによりl2⇔la⇔l3というリストができあがります．**リスト7.2**(**g**)は，指定したリストheadの後にnewを追加するコードです．また**リスト7.2**(**h**)は，リストheadの前にnewを追加するコードです．

▶ 双方向リンクでの除去

図7.4(**b**)は，l2をリンクから外す例です．これは**リスト7.2**(**i**)の__list_delで行います．**リスト7.2**(**j**)は，指定したリストentryを除去するコードです．ここでは前出〔**リスト7.2**(**c**)〕の例外を生ずるポインタを除去したリストentryのメンバに書き込んでいます．これにより，除去後にこのentryにアクセスしたとき，例外が発生します．

リスト7.2
双方向リンク

＜include/linux/kernel.h内で定義＞
構造体のメンバのポインタからその構造体のアドレスを得る
ptr　　メンバのポインタ
type　　構造体名
member　構造体のメンバ

```
#define container_of(ptr, type, member) ({                      ¥
        const typeof( ((type *)0)->member ) *__mptr = (ptr);    ¥
        (type *)( (char *)__mptr - offsetof(type,member) );})
```

(**a**)

＜include/linux/list.h内で定義＞
構造体のメンバのポインタ値からその構造体のアドレスを得る
ptr　　メンバのポインタ
type　　構造体名
member　構造体のメンバ

```
#define  list_entry(ptr, type, member) container_of(ptr, type, member)
```

(**b**)

＜include/linux/list.h内で定義＞
ページ例外を発生する．NULLでないポインタ

```
#define LIST_POISON1  ((void *) 0x00100100)
#define LIST_POISON2  ((void *) 0x00200200)
```

(**c**)

list_head構造体

```
struct list_head {
       struct list_head *next, *prev;
};
```

(**d**)

list_head構造体の定義と初期化

```
#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) struct list_head name = LIST_HEAD_INIT(name)

#define INIT_LIST_HEAD(ptr) do { ¥
        (ptr)->next = (ptr); (ptr)->prev = (ptr); ¥
} while (0)
```

(**e**)

__list_addリストの追加（内部関数）

```
static inline void __list_add(struct list_head *new,      ……追加するリスト
                              struct list_head *prev,     ……前のリスト
                              struct list_head *next)     ……後のリスト
{
       next->prev = new;        --------- ①
       new->next = next;        --------- ②   図7.4(a)のステップ
       new->prev = prev;        --------- ③
       prev->next = new;        --------- ④
}
```

(**f**)

リスト7.2
双方向リンク
（つづき）

```
list_add        リストの追加(後に)
new             追加するリストのポインタ
head            追加する場所のポインタ
headの後にnewを追加

static inline void list_add(struct list_head *new, struct list_head *head)
{
        __list_add(new, head, head->next);
}
```

(g)

```
list_add_tail   リストの追加(前に)
new             追加するリストのポインタ
head            追加する場所のポインタ
headの前にnewを追加する

static inline void list_add_tail(struct list_head *new, struct list_head *head)
{
        __list_add(new, head->prev, head);
}
```

(h)

```
__list_delリストの除去(内部関数)

前後のリストのポインタにより間のリストを除去する

static inline void __list_del(struct list_head * prev, struct list_head * next)
{
        next->prev = prev;  --------- ①  } 図7.4(b)のステップ
        prev->next = next;  --------- ②  }
}
```

(i)

```
list_del        リストのポインタで指定されたリストを除去する
entry           除去するリストのポインタ

static inline void list_del(struct list_head *entry)
{
        __list_del(entry->prev, entry->next);
        entry->next = LIST_POISON1;  } 除去したリストの
        entry->prev = LIST_POISON2;  } リンクを無効にする
}
```

(j)

```
list_move_tail      リストの移動
list                移動するリストのポインタ
head                移動先のポインタ(この後に接続される)

static inline void list_move_tail(struct list_head *list,
                                  struct list_head *head)
{
         __list_del(list->prev, list->next);
         list_add_tail(list, head);
}
```

(k)

```
list_empty      リストが空かどうかテストする
head            テストするリストのポインタ

static inline int list_empty(const struct list_head *head)
{
        return head->next == head;
}
```

(l)

リスト7.2
双方向リンク
(つづき)

```
list_for_each     リストをたどる(順方向)
pos               ループのための変数
head              開始リストのポインタ

 #define list_for_each(pos, head) ¥
    for (pos = (head)->next, prefetch(pos->next); pos != (head); ¥
             pos = pos->next, prefetch(pos->next))

利用方法は
list_for_each(pos,head)
        {
処理
  }
のようにする
```

(**m**)

```
list_for_each_prev   リストをたどる（逆方向）
pos                  ループのための変数
head                 開始するリストのポインタ

#define list_for_each_prev(pos, head) ¥
    for (pos = (head)->prev, prefetch(pos->prev); pos != (head); ¥
             pos = pos->prev, prefetch(pos->prev))
```

(**n**)

▶ リストの移動

リスト7.2(**k**)は，リンクの再接続を行うコードです．

listで指定されるリストのリンクがheadで指定されるリンクの後に再接続されます．これは**リスト7.2**(**i**)の__list_delと**リスト7.2**(**h**)のlist_add_tailによって実現されます．

▶ そのほかのリストの操作

リスト7.2(**l**)は，リストが空かどうか調べるコードです．headで指定されたlist_headのメンバnextが自分自身(head)である場合，空であることを示します．**リスト7.2**(**m**)はheadで指定されたリストから順に(前方向)にリストをたどりながら処理をするコードです．pos変数が指定されているのは，このコードを再帰的(リエントラント)に利用するためと，処理内でリストを扱えるようにするためです．**リスト7.2**(**n**)は，逆方向にリストをたどりながら処理を行うコードです．

7.2　スピンロック

マルチプロセッサ〔ここではSMP(シンメトリック・マルチプロセッサ)を仮定する〕環境では，複数のコードが異なったCPUで実行されます．OS内部では同時に特定のコードが複数のCPUで実行をされては困ることがあります．

ただ1個のCPUのみがあるコードを実行することを保証するのがスピンロックの役割です．基本的には，あるCPUが使用中のときは，ほかのCPUをループさせて待たせておくというものです．セマフォの場合と異なり，このループ実行中にスケジューリングを受けることはありません．長時間のループになる可能性があるケースでは，このループを分割して途中でスケジューリングを可能にできます．

スピンロックのロック変数は，構造体内のメンバとして定義されます．**リスト7.3**(**a**)のspinlock_t型で宣言されます．lockメンバはlong変数ですが，実際はその1バイトが用いられます．

● スピンロック関数

リスト7.3(**b**)は，スピンロックを行うspin_lock関数のコードです．実際のスピンロックは，**リスト7.3**(**f**)のコードで実現されます．**リスト7.3**(**c**)に示したように，多くの場所で論理的には無意味な関数likelyやunlikelyが呼ばれています．これは，コンパイラのコード生成の手助けとして用いる関数です．マシン・コードに落ちたとき，ブランチ命令の飛び先をどちらにするかを決めます．つまり，確率的に低い発生率のほうをブ