

【Kernel】

【Sigma0】

【RMGR】

crt0.S

startup( ) startup.c:569

sigma0をロード  
kernelをロード  
kernelに制御を渡す。

startup.S

kernel/src/x86/i686/startup.S

CALL init\_paging( ) kernel/src/x86/init.c:232

\_kmem\_start, \_kmem\_end を設定  
PageDirectory とPageTable (1個) を割りつける  
PageDirectory の#0エンタリを 0番地の4MB SuperPage に設定  
PageDirectory の3.75G+1Mのエンタリに該PageTable を登録  
PageDirectory ->該PageTable の3.75G+1Mから18ページ分を登録  
PageDirectory の3.75G+4Mから 240M 分のエンタリを4MB SuperPage に設定  
ビデオメモリを登録  
enable\_super\_page()  
set\_current\_pagetable(pdir\_root)  
enable\_paged\_mode()  
kernel\_ptdir\_root = phys\_to\_virt(pdur=root);  
ここから仮想メモリが有効になる。

JUMP init( );

init( )

kernel/src/init.c

init\_arch\_1( )

init\_irqs( )

全IRQsをマスク  
PIC-1 に対して  
初期設定命令 ICW1, ICW2, ICW3, ICW4 を出す  
操作命令 OCW2 を出す  
PIC-2 に対して  
初期設定命令 ICW1, ICW2, ICW3, ICW4 を出す  
操作命令 OCW2 を出す  
PIC-1 に対して OCW3 を出す  
PIC-2 に対して OCW3 を出す  
タイマIRQ(int 8)をEnable  
Reset IS-bit-0  
Enable NMI

calc\_processor\_speed()

RTC を512Hz に設定

Intrラインをリセット



```

変数 free_mem = kip -> main_mem_low;
kernel_mem = kip -> main_mem_high;

```

page\_arrayを初期設定 (Kernel空間, sigma0空間等を予約済にする)

LOOP

```

l4_ipc_wait(&client, 0, &dw0, &dw1, .... );
msg = 2; // Fpage mapを意味する

```

```

IF client がカーネルスレッド
THEN

```

```

IF dw0== 0xFFFFFFFF
THEN // カーネルページの要求
page_arrayテーブルを上位アドレスから探す.
IF 空きページ有り
THEN dw0 = そのアドレス;
dw1 = dw0 | 1-page | Grant;
page_arrayテーブル更新;
//⇒空きページをカーネルページ(4K)としてGrant する。
ELSE msg = dw0 = dw1 = 0;
FI;

```

all '1'	00
---------	----

```

ELSIF dw0== 0x00000001
THEN
IF dw1 & 0xFF == 0
THEN //
msg=0; dw0=0x100; //Num of pages for kernel use.
//⇒カーネルが使えるページ数を返す。
ELSE
msg = dw0 = dw1 = 0;
FI;

```

00 ----- 001	
	0

```

ELSE
不当メッセージ
FI;

```

```

ELSE // client はユーザスレッド

```

```

IF dw0== 0xFFFFFFFF
THEN // Map a 4K page writeable
page_arrayテーブルを下位アドレスから探す.
IF 空きページ有り
THEN dw0 =そのアドレス;
dw1 = dw0 | 1-page | Map ;
page_array テーブルに client を登録;
// ⇒空きページをカーネルページ(4K)としてMap する。
ELSE msg = dw0 = dw1 = 0;
FI;

```

11-----1100	
-------------	--

```

ELSIF dw0== 0x00000001
THEN
IF dw1 & 0xFF == 1
THEN // カーネルinfoページ要求
msg = 0; dw0 = 0;
dw1=カーネルinfoページのアドレス | 4K-page | Map ;
//⇒カーネルinfoページ(4K)をMap する。
ELSE
msg = dw0 = dw1 = 0;
FI;

```

00 -----001	
	0-01

```

ELSIF dw0 == 0 OR 0x00000002 <= dw0 <= 0x40000000
THEN

```

0G<= <=1G	1
	SP

```

IF (dw0 & 0X01) && (dw1 & 0XFF)==SuperPageBit << 2)
THEN // Map Superpage writable and uncacheble
adr = dw0 & SUPERPAGE_MASK;

```

```

page_arrayテーブルに adr からSuperpage サイズ分の空きがあるか?
IF 空きページ有り
    THEN dw0 =adr;
        dw1 = dw0 | SuperPage | Map ;
        page_array テーブルに client を登録;
        //⇒要求された番地にスーパーページ(4M)をMap する。
    ELSE
        msg = dw0 = dw1 = 0;
FI;

ELSE
    IF 該当ページが割り付け済
    THEN msg = dw0 = dw1 = 0;
    ELSE page_arrayテーブルに client を登録;
        dw0 &= PAGE_MASK;
        dw1 = dw0 | 4K-page | Map ;
        dw2 = 0;
        // ⇒空きページを要求された番地に(4K)Map する。
    FI
FI

ELSIF 0x40000004 <= dw0 <= 0xC0000000
THEN // Map 4M Superpage
    dw0 = (dw0 & SUPERPAGE_MASK) + 0X40000000;
    dw1 = dw0 | Superpage | Map;
    //⇒要求値 + 1G番地のスーパーページ(4M)をMap する。
ELSE
FI;

FI;

I4_ipc_reply(client, msg, dw0, dw1, .... );
ENDLOOP;

```

1G<	<=3G
-----	------

### 【RMGRのブート】

```

init( )      RMGRの init.c:51

printf("RMGR: High there!¥n");
init_globals()

init_memmap()

FOR ( 4M <= adrs <128M AND メモリ有り)
    I4_ipc_call(sigma0, 0, adrs|1, 22<<2, ...);
    // 4MB から ..MB までSuper page(4MB) をSigma0からmap
END FOR

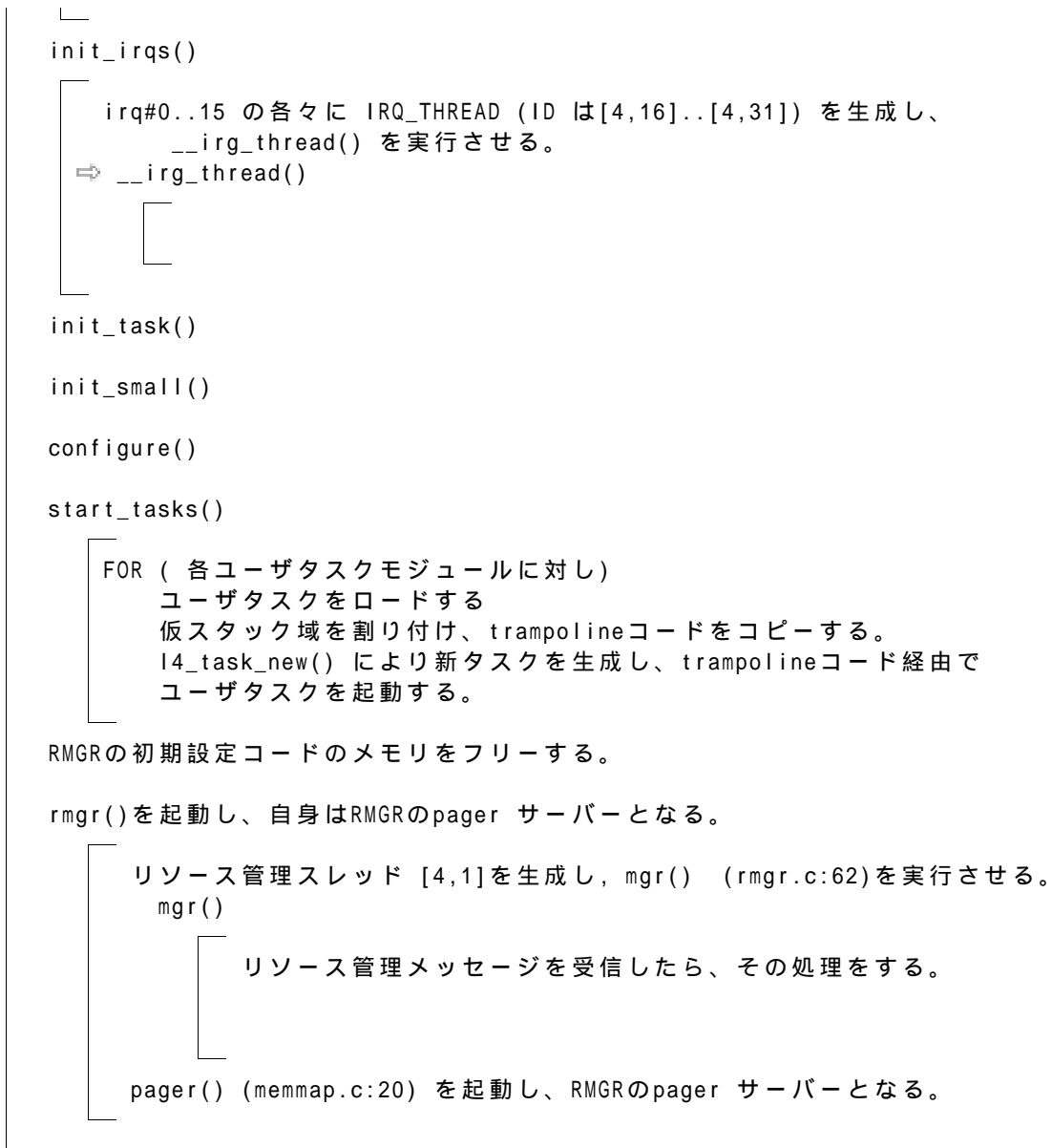
FOR ( ; ; )
    I4_ipc_call(sigma0, 0, 0xFFFFFFFFc, 0, ...);
    // 標準page(4KB) をSigma0からmap
END FOR

Adapter space (640K..1M)

RMGR自身のページ

Kernel info page

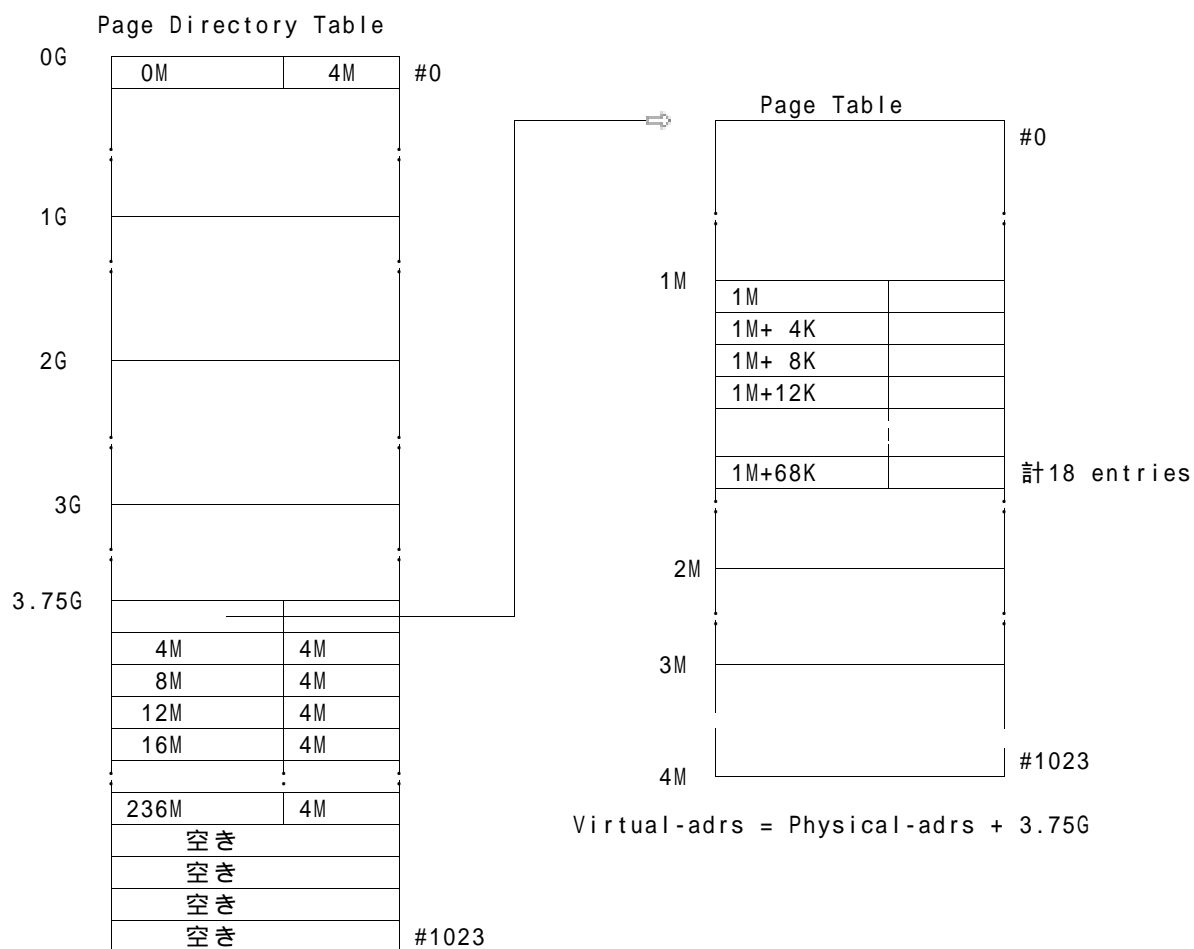
```



【 GDTの設定内容】

		src/x86/init.c:56
#0	unused	
#1	Kernel Code Seg	base-adrs=0 segment-limit= 4G-1 G=1, D/B=1, AVL=0, P=1, DPL=0, S=1 Type= 実行 R/W可能
#2	Kernel Data Seg	base-adrs=0 segment-limit= 4G-1 G=1, D/B=1, AVL=0, P=1, DPL=0, S=1 Type= データ R/W可能
#3	User Code Seg	base-adrs=0 segment-limit= 4G-1 G=1, D/B=1, AVL=0, P=1, DPL=3, S=1 Type= 実行 R/W可能
#4	User Data Seg	base-adrs=0 segment-limit= 4G-1 G=1, D/B=1, AVL=0, P=1, DPL=3, S=1 Type= データ R/W可能
#5	TSS Seg	base-adrs=_tssのアドレス segment-limit= 103 G=0, D/B=0, AVL=1, P=1, DPL=0, S=0 Type= 実行、Rのみ ⇒ TSS の 10-bit-partは使わない

## 【Page Directory TableとPage Table】



○ I4-ka/kernel/linker/x86.lds

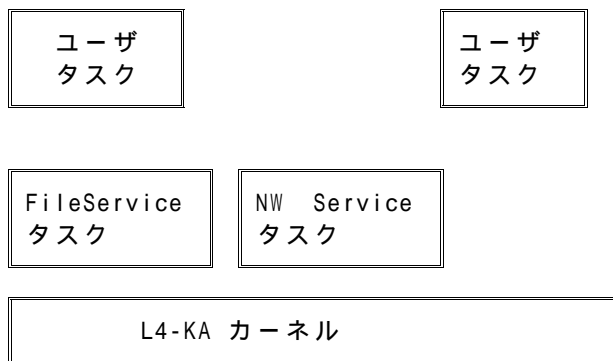
○ I4-ka/kernel/include/x86/config.h

```
TCB_AREA          3.5G
TCB_AREA_SIZE     16M
MEM_COPYAREA1     3.5G + 16M
MEM_COPYAREA2     3.5G + 16M + 8M
MEM_COPYAREA_END  3.5G + 16M + 16M
```

```
KERNEL_PHYS       1M
KERNEL_OFFSET     3.75G
KERNEL_VIRT       3.75G + 1M
KERNEL_SIZE       4K * 18
```

# 実 施 計 画

## § マルチサーバOS



Cf. ChacmOS, SawMill

サービス毎のタスク  
独立空間  
ユーザモードじ実行  
ハードウェア駆動  
頑強なガード

### サービスタスク

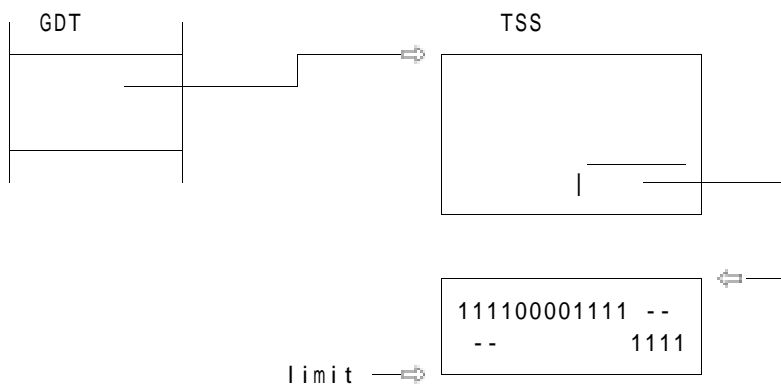


サービスタスクはハードウェア の駆動ができること  
IRQ handling  
IO空間のアクセス — in, out 命令  
Mapped-IO の利用  
頑強なガードの基に実現したい。

### IO空間のアクセス制御

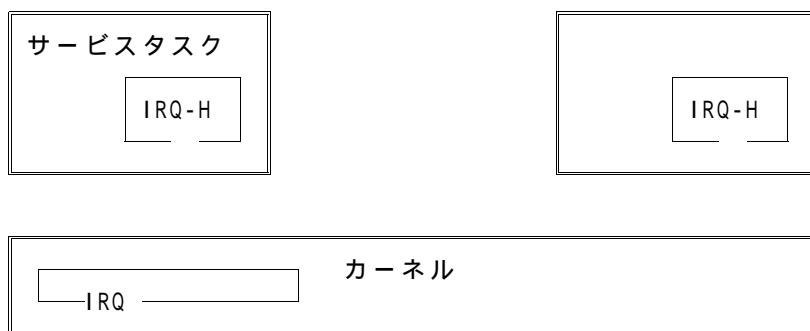
○実行モードによる制御：IO空間のアクセスをカーネルモードのみに許す  
⇒サービスタスクをユーザモードで動作させるには、こちらは使えない

○I<sub>x</sub>86 の TSSテーブルに付随するIO-bitmap の利用

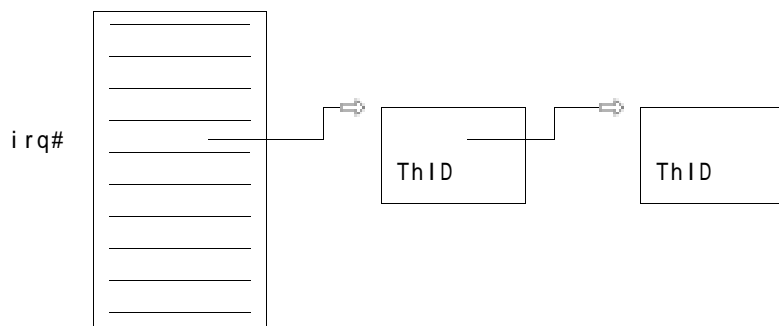


タスク毎に TSSを設ける  
TSS は 1個で、IO-bitmap をタスク毎に替える

## IRQ のカーネルへの登録



interrupt\_ownerテーブル



## インプリメント

- 簡単なデバイスがあれば、それを使える
- HD と NW を使う
  - OSKitのソースを使う —— Glueコードが付いているが、中身は複雑
  - Linuxのソースを使う —— Glueコード相当を追加する必要がある。

## 割り込み処理モデル

- 割り込み処理プロシージャ —— 世の中の大部分
- 割り込み処理タスク —— Ix86 は、このための機構もあるが、複雑

## ChacmOS

マルチサーバOSの適切な例題  
IDL: kernel call を ipcに展開  
タスクの起動のための trampoline 機構  
プログラムのロード

## Fiasco + OSKit からの学習

- Sigma0
- Rmgr
- OSKit support

## § 分散処理