Linux 記憶體點滴使用者行程記憶體空間

經常使用top命令了解進程信息，其中包括內存方面的信息。命令top幫助文檔是這麼解釋各個字段的。

VIRT , Virtual Image (kb)

RES, Resident size (kb)

SHR, Shared Mem size (kb)

%MEM, Memory usage(kb)

SWAP, Swapped size (kb)

CODE, Code size (kb)

DATA, Data+Stack size (kb)

nFLT, Page Fault count

nDRT, Dirty Pages count

儘管有註釋，但依然感覺有些晦澀，不知所指何意？

進程內存空間

正在運行的程序，叫進程。每個進程都有完全屬於自己的，獨立的，不被干擾的內存空間。此空間，被分成幾個段(Segment),分別是Text, Data, BSS, Heap, Stack。用戶進程內存空間，也是系統內核分配給該進程的VM(虛擬內存)，但並不表示這個進程佔用了這麼多的RAM(物理內存)。這個空間有多大？命令top輸出的VIRT值告訴了我們各個進程內存空間的大小（進程內存空間隨著程序的執行會增大或者縮小）。你還可以通過/proc//maps，或者pmap –d 了解某個進程內存空間都分佈,比如:

#cat /proc/1449/maps

…

0012e000 - 002a4000 r - xp 00000000 08 : 07 3539877 / lib / i386 - linux - gnu / libc - 2.13. so

002a4000 - 002a6000 r -- p 00176000 08 : 07 3539877 / lib / i386 - linux - gnu / libc - 2.13. so

002a6000 - 002a7000 rw - p 00178000 08 : 07 3539877 / lib / i386 - linux - gnu / libc - 2.13. so

002a7000 - 002aa000 rw - p 00000000 00 : 00 0

…

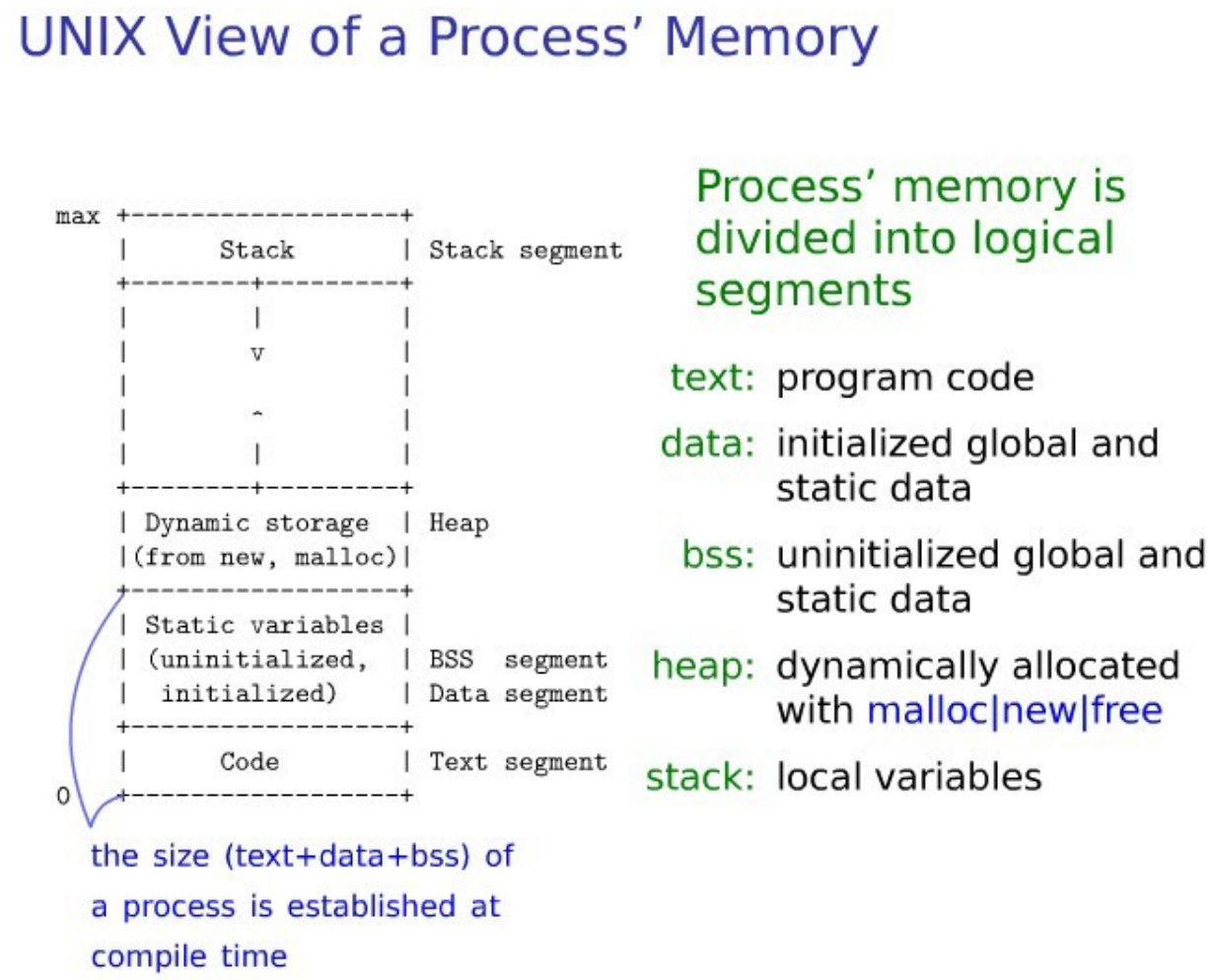
08048000 - 0875b000 r - xp 00000000 08 : 07 4072287 / usr / local / mysql / libexec / mysqld

0875b000 - 0875d000 r -- p 00712000 08 : 07 4072287 / usr / local / mysql / libexec / mysqld

0875d000 - 087aa000 rw - p 00714000 08 : 07 4072287 / usr / local / mysql / libexec / mysqld

…

PS :線性地址，訪問權限, offset ,設備號，inode，映射文件



VM分配與釋放

“內存總是被進程佔用”，這句話換過來可以這麼理解：進程總是需要內存。當fork()或者exec()一個進程的時候，系統內核就會分配一定量的VM給進程，作為進程的內存空間，大小由BSS段，Data段的已定義的全局變量、靜態變量、Text段中的字符直接量、程序本身的內存映像等，還有Stack段的局部變量決定。當然，還可以通過malloc()等函數動態分配內存,向上擴大heap。

動態分配與靜態分配，二者最大的區別在於:1. 直到Run-Time的時候，執行動態分配，而在compile-time的時候，就已經決定好了分配多少Text+Data+BSS+Stack。2.通過malloc()動態分配的內存，需要程序員手工調用free()釋放內存，否則容易導致內存洩露，而靜態分配的內存則在進程執行結束後系統釋放(Text, Data), 但Stack段中的數據很短暫，函數退出立即被銷毀。
我們使用幾個示例小程序，加深理解

```
/* @filename: example-2.c */
#include <stdio.h>

int main ( int argc , char * argv [ ] )
{
    char arr [ ] = "hello world" ;/* Stack段，rw--- */
    char * p = "hello world" ;              /* Text段，字符串直接量, rx-- */
    arr [ 1 ] = 'l' ;
    * ( ++ p ) = 'l' ;        /*出錯了,Text段不能write */
    return 0 ;
}
```

PS :變量p，它在Stack段，但它所指的”hello world”是一個字符串直接量，放在Text段。

```
/* @filename:example_2_2.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char * get_str_1 ( )
{
    char str [ ] = "hello world" ;
    return str ;
}
```

```
char * get_str_2 ( )
{
    char * str = "hello world" ;
    return str ;
}

char * get_str_3 ( )
{
    char tmp [ ] = "hello world" ;
    char * str ;
    str = ( char * ) malloc ( 12 * sizeof ( char ) ) ;
    memcpy ( str , tmp , 12 ) ;
    return str ;
}

int main ( int argc , char * argv [ ] )
{
    char * str_1 = get_str_1 ( ) ;        //出錯了，Stack段中的數據在函數退出時就銷毀了
    char * str_2 = get_str_2 ( ) ;        //正確，指向Text段中的字符直接量，退出程序後才會回收
    char * str_3 = get_str_3 ( ) ;        //正確，指向Heap段中的數據，還沒free()
    printf ( "%s \n " , str_1 ) ;
    printf ( "%s \n " , str_2 ) ;
    printf ( "%s \n " , str_3 ) ;
    if ( str_3 != NULL )
    {
        free ( str_3 ) ;
        str_3 = NULL ;
```

```
    }
    return 0 ;
}
```

PS :函數get_str_1 ( )返回Stack段數據，編譯時會報錯。Heap中的數據，如果不用了，應該儘早釋放free ( )。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

char data_var   = '1' ;
char * mem_killer ( )
{
   char * p ;
   p = ( char * ) malloc ( 1024 * 1024 * 4 ) ;
   memset ( p , ' \0 ' , 1024 * 1024 * 4 ) ;
   p = & data_var ;      //危險，內存洩露
   return p ;
}

int main ( int argc , char * argv [ ] )
{
    char * p ;
    for ( ;; )
    {
        p = mem_killer ( ) ; //函數中malloc()分配的內存沒辦法free()
        printf ( "%c \ n " , * p ) ;
```

```
        sleep ( 20 ) ;
    }
    return 0 ;
}
```

PS :使用malloc ( )，特別要留意heap段中的內存不用時，儘早手工free ( )。通過top輸出的VIRT和RES兩值來觀察進程佔用VM和RAM大小。

本節結束之前，介紹工具size。因為Text, BSS, Data段在編譯時已經決定了進程將佔用多少VM。可以通過size，知道這些信息。

```
# gcc example_2_3.c -o example_2_3
# size example_2_3
text data bss dec hex filename
1403 272 8 1683 693 example_2_3
```

Stack vs. Heap

Stack	Heap
compile-time allocation	run-time allocation
auto clean-up	you clean-up
inflexible	flexible
smaller	bigger
quicker	slower

How large is the ...

stack: `~$ ulimit -s`

heap: could be as large as your virtual memory

malloc()

編碼人員在編寫程序之際，時常要處理變化數據，無法預料要處理的數據集變化是否大（phper可能難以理解），所以除了變量之外，還需要動態分配內存。GNU libc庫提供了二個內存分配函數,分別是malloc()和calloc()。調用malloc(size_t size)函數分配內存成功，總會分配size字節VM（再次強調不是RAM），並返回一個指向剛才所分配內存區域的開端地址。分配的內存會為進程一直保留著，直到你顯示地調用free()釋放它（當然，整個進程結束，靜態和動態分配的內存都會被系統回收）。開發人員有責任儘早將動態分配的內存釋放回系統。記住一句話:儘早free() !

我們來看看，malloc()小示例。

```
/* @filename:example_2_4.c */
#include <stdio.h>
#include <stdlib.h>

int main ( int argc , char * argv [ ] )
{
   char * p_4kb , * p_128kb , * p_300kb ;
   if ( ( p_4kb = malloc ( 4 * 1024 ) ) != NULL )
   {
      free ( p_4kb ) ;
   }
   if ( ( p_128kb = malloc ( 128 * 1024 ) ) != NULL )
   {
      free ( p_128kb ) ;
   }
   if ( ( p_300kb = malloc ( 300 * 1024 ) ) != NULL )
   {
      free ( p_300kb ) ;
```

```
    }
    return 0 ;
}
#gcc example_2_4. c –o example_2_4
#strace –t ./example_2_4
…
00 : 02 : 53 brk ( 0 )                    = 0x8f58000
00 : 02 : 53 brk ( 0x8f7a000 )            = 0x8f7a000
00 : 02 : 53 brk ( 0x8f79000 )            = 0x8f79000
00 : 02 : 53 mmap2 ( NULL , 311296 , PROT_READ | PROT_WRITE , MAP_PRIVATE |
MAP_ANONYMOUS , - 1 , 0 ) = 0xb772d000
00 : 02 : 53 munmap ( 0xb772d000 , 311296 )    = 0
…
```

PS :系統調用brk (0)取得當前堆的地址，也稱為斷點。

通過跟踪系統內核調用，可見glibc函數malloc()總是通過brk()或mmap()系統調用來滿足內存分配需求。函數malloc()，根據不同大小內存要求來選擇brk()，還是mmap()， 128Kbytes是臨界值。小塊內存(<=128kbytes)，會調用brk()，它將數據段的最高地址往更高處推（堆從底部向上增長）。大塊內存，則使用mmap()進行匿名映射(設置標誌MAP_ANONYMOUS)來分配內存，與堆無關，在堆之外。這樣做是有道理的，試想：如果大塊內存，也調用brk()，則容易被小塊內存釘住，必竟用大塊內存不是很頻繁;反過來，小塊內存分配更為頻繁得多，如果也使用mmap()，頻繁的創建內存映射會導致更多的開銷，還有一點就是，內存映射的大小要求必須是“頁”（單位，內存頁面大小，默認4Kbytes或8Kbytes）的倍數,如果只是為了”hello world”這樣小數據就映射一“頁”內存，那實在是太浪費了。

跟malloc()一樣，釋放內存函數free()，也會根據內存大小，選擇使用brk()將斷點往低處回推，或者選擇調用munmap()解除映射。有一點需要注意：並不是每次調用free()小塊內存，都會馬上調用brk()，即堆並不會在每次內存被釋放後就被縮減，而是會被glibc保留給下次malloc ()使用(必竟小塊內存分配較為頻繁)，直到glibc發現堆空閒大小顯著大於內存分配所需數量時，則會調用brk()。但每次free()大塊內存，都會調用munmap()解除映射。下面是二張malloc()小塊內

存和大塊內存的示例圖。

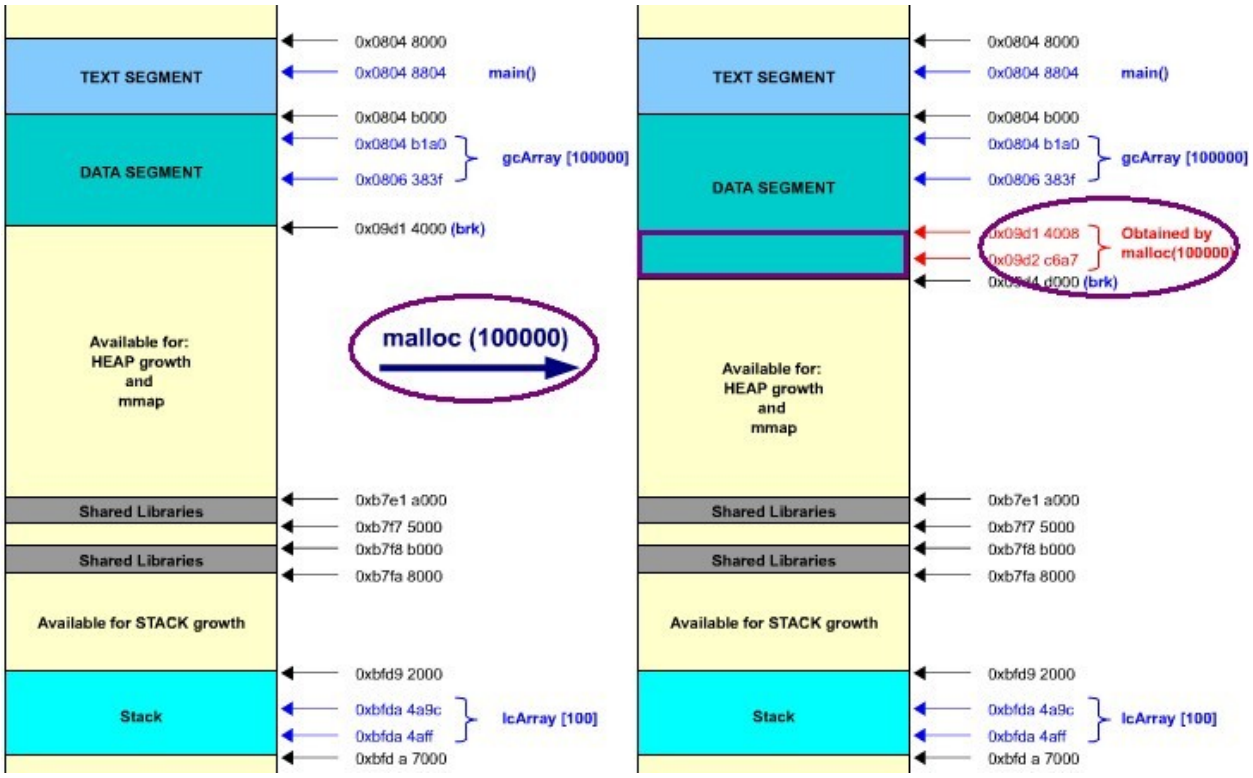
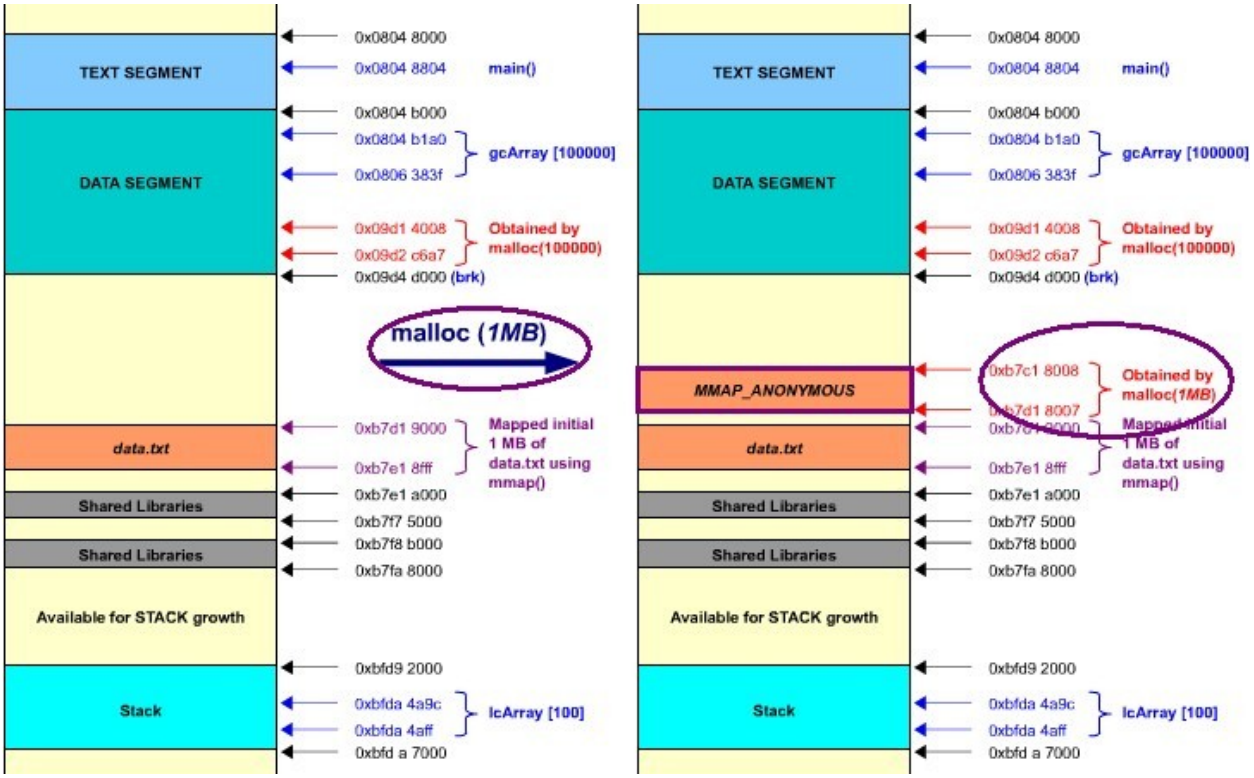


示意圖:函數malloc(100000)，小於128kbytes，往高處推(heap->)。留意紫圈標註



示意圖：函數malloc(1024*1024)，大於128kbytes，在heap與stack之間。留意紫圈。PS:圖中的Data Segment泛指BSS, Data, Heap。有些文檔有說明：數據段有三個子區域，分別是BSS, Data, Heap。

缺頁異常(Fault Page)

每次調用malloc()，系統都只是給進程分配線性地址（VM），並沒有隨即分配頁框(RAM)。系統盡量將分配頁框的工作推遲到最後一刻—用到時缺頁異常處理。這種頁框按需延遲分配策略最大好處之一：充分有效地善用系統稀缺資源RAM。

當指針引用的內存頁沒有駐留在RAM中，即在RAM找不到與之對應的頁框，則會發生缺頁異常(對進程來說是透明的)，內核便陷入缺頁異常處理。發生缺頁異常有幾種情況：1.只分配了線性地址，並沒有分配頁框，常發生在第一次訪問某內存頁。2.已經分配了頁框，但頁框被回收，換出至磁盤(交換區)。3.引用的內存頁，在進程空間之外，不屬於該進程，可能已被free()。我們使用一段偽代碼來大致了解缺頁異常。

/* @filename: example_2_5.c */

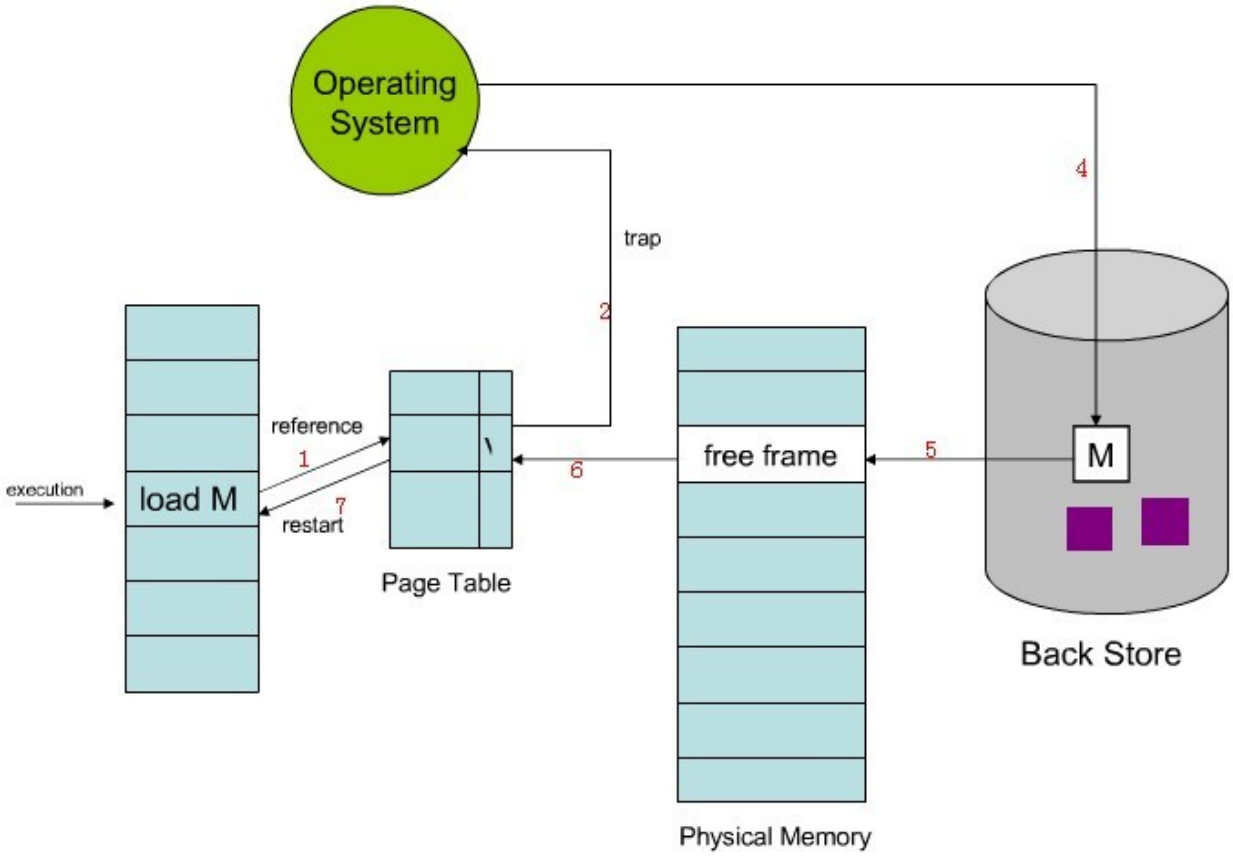
…

```
demo ( )
{
   char * p ;
   //分配了100Kbytes線性地址
   if ( ( p = malloc ( 1024 * 100 ) ) != NULL )  // L0
   {
      * p = 't' ;    // L1
        … //過去了很長一段時間，不管系統忙否，長久不用的頁框都有可能被回收
        * p = 'm' ;          // L2
        p [ 4096 ] = 'p' ;   // L3
        …
        free ( p ) ;  //L4
        if ( p == NULL )
        {
                * p = 'l' ; // L5
        }
   }
}
…
```

- L0，函數malloc()通過brk()給進程分配了100Kbytes的線性地址區域(VM).然而，系統並沒有隨即分配頁框(RAM)。即此時，進程沒有佔用100Kbytes的物理內存。這也表明了，你時常在使用top的時候VIRT值增大，而RES值卻不變的原因。
- L1，通過*p引用了100Kbytes的第一頁(4Kbytes)。因為是第一次引用此頁，在RAM中找不到與之相對應的頁框。發生缺頁異常（對於進程而言缺頁異常是透明的），系統靈敏地捕獲這一異常，進入缺頁異常處理階段：接下來，系統會分配一個頁框(RAM)映射給它。我們把這種情況(被訪問的頁還沒有被放在任何一個頁框中，內核分配一新的頁框並適當初始化來滿足調用請求)，也稱為Demand Paging。
- L2，過了很長一段時間，通過*p再次引用100Kbytes的第一頁。若係統在RAM找不到它映射的頁框(可能交換至磁盤了)。發生缺頁異常，並被系統捕獲進入缺頁異常處理。接下來，系統則會分配一頁頁框(RAM)，找到備份在磁盤的那“頁”，並將它換入內存(其實因為換入操作比較昂貴，所以不總是只換入一頁，而是預換入多頁。這也表明某些文檔說：”vmstat某時出現不少si並不能意味著物理內存不足”)。凡是類似這種會迫使進程去睡眠（很可能是由於當前磁盤數據填充至頁框(RAM)所花的

時間）,阻塞當前進程的缺頁異常處理稱為主缺頁(major falut)，也稱為大缺頁(參見下圖)。相反，不會阻塞進程的缺頁，稱為次缺頁(minor fault)，也稱為小缺面。

- L3，引用了100Kbytes的第二頁。參見第一次訪問100Kbytes第一頁, Demand Paging。
- L4，釋放了內存：線性地址區域被刪除，頁框也被釋放。
- L5，再次通過*p引用內存頁，已被free()了(用戶進程本身並不知道)。發生缺頁異常，缺面異常處理程序會檢查出這個缺頁不在進程內存空間之內。對待這種編程錯誤引起的缺頁異常，系統會殺掉這個進程，並且報告著名的段錯誤(Segmentation fault)。



主缺頁異常處理過程示意圖,參見Page Fault Handling

頁框回收PFRA

隨著網絡並髮用戶數量增多，進程數量越來越多(比如一般守護進程會fork()子進程來處理用戶請求)，缺頁異常也就更頻繁，需要緩存更多的磁盤數據(參考下篇OS Page Cache)，RAM也就越來越緊少。為了保證有夠用的頁框供給缺頁異常處理，Linux有一套自己的做法，稱為PFRA。

PFRA總會從用戶態進內存程空間和頁面緩存中，“竊取”頁框滿足供給。所謂”竊取”，指的是：將用戶進程內存空間對應佔用的頁框中的數據swap out至磁盤(稱為交換區)，或者將OS頁面緩存中的內存頁（還有用戶進程mmap()的內存頁）flush(同步fsync())至磁盤設備。PS:如果你觀察到因為RAM不足導致系統病態式般慢，通常都是因為缺頁異常處理，以及PFRA在”盜頁”。我們從以下幾個方面了解PFRA。

候選頁框

- ：找出哪些頁框是可以被回收？進程內存空間佔用的頁框，比如數據段中的頁(Heap, Data)，還有在Heap與Stack之間的匿名映射頁(比如由malloc()分配的大內存)。但不包括Stack段中的頁。
- 進程空間mmap()的內存頁，有映射文件，非匿名映射。
- 緩存在頁面緩存中Buffer/Cache佔用的頁框。也稱OS Page Cache。

頁框回收策略

- ：確定了要回收的頁框，就要進一步確定先回收哪些候選頁框盡量先回收頁面緩存中的Buffer/Cache。其次再回收內存空間佔用的頁框。
- 進程空間佔用的頁框，要是沒有被鎖定，都可以回收。所以，當某進程睡眠久了，佔用的頁框會逐漸地交換出去至交換區。
- 使收LRU置換算法，將那些久而未用的頁框優先被回收。這種被放在LRU的unused鍊錶的頁，常被認為接下來也不太可能會被引用。
- 相對回收Buffer/Cache而言，回收進程內存頁，昂貴很多。所以，Linux默認只有swap_tendency(交換傾向值)值不小於100時，才會選擇換出進程佔用的RES。其實交換傾向值描述的是：系統越忙，且RES都被進程佔用了，Buffer/Cache只佔了一點點的時候，才開始回收進程佔用頁框。PS:這正表明了，某些DBA提議將MySQL InnoDB服務器vm.swappiness值設置為0，以此讓InnoDB Buffer Pool數據在RES呆得更久。
- 如果實在是沒有頁框可回收，PFRA使出最狠一招，殺掉一個用戶態進程，並釋放這些被佔的頁框。當然，這個被殺的進程不是胡亂選的，至少應該是佔用較多頁框，運行優選級低，且不是root用戶的進程。

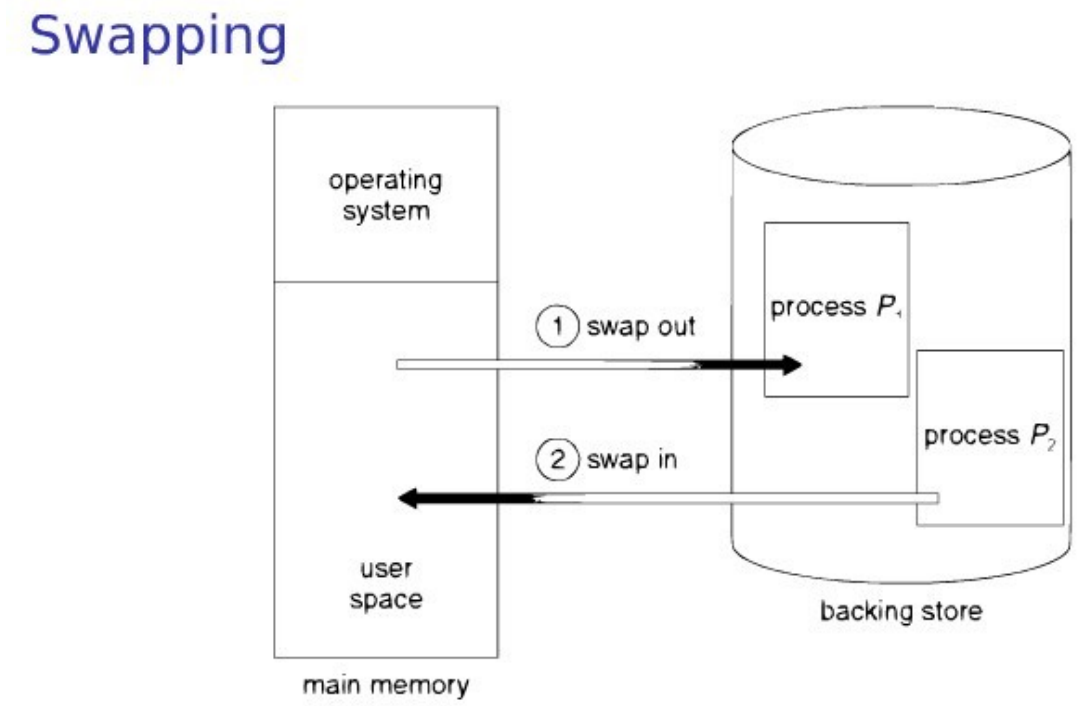
激活回收頁框

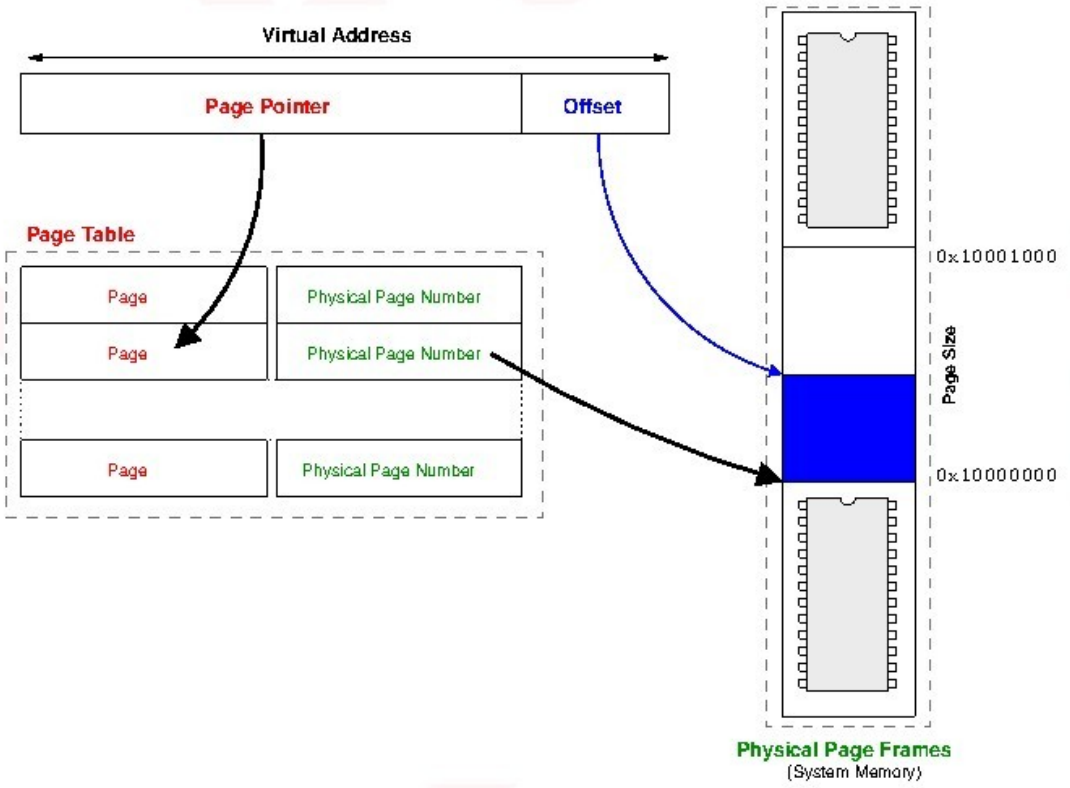
- ：什麼時候會回收頁框?緊急回收。系統內核發現沒有夠用的頁框分配，供給讀文件和內存缺頁處理的時候，系統內核開始”緊急回收頁框”。喚醒pdflush內核線程，先將1024頁臟頁從頁面緩存寫回磁盤。然後開始回收32頁框，若反復回收13次，還收不齊32頁框，則發狠殺一個進程。
- 週期性回收。在緊急回收之前，PFRA還會喚醒內核線程kswapd。為了避免更多的“緊急回收”，當發現空閒頁框數量低於設置的警告值時，內核線程kswapd就會被喚醒，回收頁框。直到空閒的頁框的數量達到設定的安全值。PS:當RES資源緊張的時候，你可以通過ps命令看到更多的kswapd線程被喚醒。

- OOM。在高峰時期，RES高度緊張的時候，kswapd持續回收的頁框供不應求，直到進入”緊急回收”，直到OOM。

Paging 和Swapping

這二個關鍵字在很多地方出現，譯過來應該是Paging（調頁），Swapping(交換)。PS:英語裡面用得多的動詞加上ing，就成了名詞，比如building。咬文嚼字，實在是太難。看二圖Swapping的大部分時間花在數據傳輸上，交換的數據也越多，意味時間開銷也隨之增加。對於進程而言，這個過程是透明的。由於RAM資源不足，PFRA會將部分匿名頁框的數據寫入到交換區(swap area)，備份之，這個動作稱為so(swap out)。等到發生內存缺頁異常的時候，缺頁異常處理程序會將交換區(磁盤)的頁面又讀回物理內存，這個動作稱為si(swap in)。每次Swapping，都有可能不只是一頁數據，不管是si，還是so。Swapping意味著磁盤操作，更新頁表等操作，這些操作開銷都不小，會阻塞用戶態進程。所以，持續飚高的si/so意味著物理內存資源是性能瓶頸。Paging，前文我們有說過Demand Paging。通過線性地址找到物理地址，找到頁框。這個過程，可以認為是Paging，對於進程來講，也是透明的。Paging意味著產生缺頁異常，也有可能是大缺頁，也就意味著浪費更多的CPU時間片資源。





總結

1.用戶進程內存空間分為5段,Text, DATA, BSS, Heap, Stack。其中Text只讀可執行，DATA全局變量和靜態變量,Heap用完就儘早free()，Stack裡面的數據是臨時的，退出函數就沒了。

2.glibc malloc()動態分配內存。使用brk()或者mmap()，128Kbytes是一個臨界值。避免內存洩露，避免野指針。

3.內核會盡量延後Demand Paging。主缺頁是昂貴的。

4.先回收Buffer/Cache佔用的頁框，然後程序佔用的頁框,使用LRU置換算法。調小vm.swappiness值可以減少Swapping,減少大缺頁。

5.更少的Paging和Swapping

6.fork()繼承父進程的地址空間，不過是只讀，使用cow技術,fork()函數特殊在於它返回二次。