這份作業是用 google 表單繳交，完整的題目參考這裡，答案請到 `https://forms.gle/67dBSpqPSETPt5Kd7` 作答，不需要繳交紙本或是額外填寫繳交表單。表單將於 2025/03/22 16:59:59 截止，逾時作答將不計分，請各位同學把握時間作答。

1 記憶體布局

大家是否有寫一個遞迴，結果一執行下去程式就馬上當掉的經驗呢？很多時候這可能是遞迴層數太多而導致 stack overflow 的問題。什麼是 stack overflow 呢？為了解釋這個概念，我們在以下順便介紹一般 C 語言程式的記憶體布局 (memory layout)。

現代電腦有許多不同的 OS (Operating System，作業系統) 可供使用，例如 Windows、MacOS 以及 Linux，不同的 OS 可能會有不同的 memory layout，由於 Linux 是一項公開原始碼的常用 OS (Sprout OJ 也在用！)，因此我們接下來將以 Linux 為主。

以 Linux 為例，一個 C 程式的 memory layout 如 Figure 1。

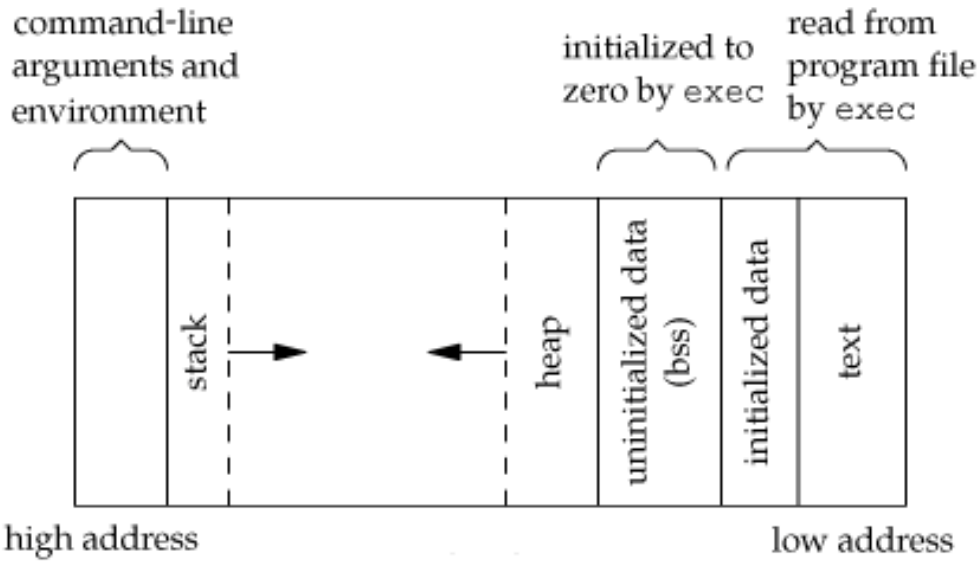


Figure 1: 一個執行中的程式在 Linux 上的記憶體布局

在圖中我們可以看到 C 程式的記憶體配置可以分成 5 個區塊 (segment)，區塊在越左邊，代表在記憶體中通常位於較高的位置 (high address, 可以想成所在記憶體的編號通常比較大，也代表著指標的值越大)。各個區塊的意義如下：

- **Text Segment**

  又稱為 code segment，一般而言簡稱為 text，基本上就是存放程式指令碼的地方。我們寫的 C 程序在經過編譯器優化、編譯之後，會轉變成只有電腦才看得懂的機器碼，這些機器碼會形成最後的執行檔的一部分，並且在程式起跑時載入到記憶體的 text

區。一般而言，text 區都會被系統設定為唯讀 (read only)，以避免被使用者不小心或者刻意修改，形成恐怖的不可預期的致命錯誤。

- **Initialized Data Segment**

  有些變數是一執行就固定位置的，例如全域變數 (global variable)、靜態變數 (static variable)，這些變數如果在程式碼中有被使用者手動初始化 (這邊的手動初始化是指宣告時就順便賦值的初始化，事後自己賦值或者 memset 等等的並不算)，就會被配置於此區塊。

- **Uninitialized Data Segment**

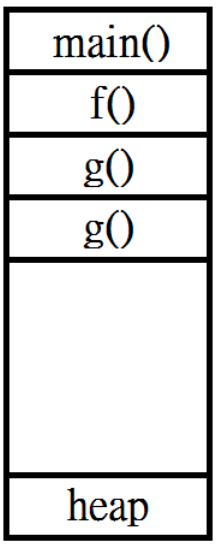
  此一區塊又被稱為 bss，意為「僅由符號起始的區塊」(block started by symbol)，在這個區塊內的變數都是使用者沒有在宣告時初始化內容的全域變數或靜態變數。值得一提的是，儘管使用者並沒有手動初始化這些變數，這些變數都會被自動初始化為 0。

- **Heap**

  當我們直接或間接使用到 malloc 時，系統動態配給的記憶體的所在區塊。這個區域通常是「往上長」的，也就是說當我們多次動態配置記憶體，新配置到的記憶體通常會出現在較高的位置，疊加在先前配置的記憶體上面。

- **Stack**

  本次的主角，原則上所有自動變數 (automatic variable)，也就是所有在 function 中宣告的非靜態變數 (注意到 main 本身就是一個 function！) 都會位於這個區塊。由於 C 語言支援函式的特性，函式內的變數並不是一開始就獨立存在的，而是進入函式並且宣告後才真實存在。每當我們多宣告了一個變數，系統就會把該變數推入到 stack 當中；而當我們呼叫 function 時，該 function 內的變數們就會疊加到 stack 的最頂端。例如我們有三個函式 main(), f(), g()，然後我們先進入 main()，接著呼叫 f()，f() 又呼叫 g()，g() 又遞迴自己一次，那麼當前的 stack 大概會長得像：



當然實際上不是真的把整個函式作為物件推入 stack，圖中的函式事實上是代表呼叫該函式時所花費的記憶體空間，除了使用者在函式內宣告的區域變數（Local variable）外，還包含一些「呼叫函式」這個動作所需的系統變數。在記憶體配置的概念上，stack 和 heap 總是反向的，例如 heap 如果是往 high address 長 (大部分如此)，stack 就會往 low address 長；heap 往 low address 長，stack 就往 high address 長，原因是因為這樣可以避免兩者在還有記憶體可用的情況下就因為預留空間不足而相撞的問題。

瞭解記憶體的架構之後，考慮一個問題：如果 stack 的深度沒有上限，那麼當使用者不小心寫出了一個無窮遞迴時，程序會發生什麼事情？這程序會一直把新的變數推入 stack，直到把記憶體用光為止。這顯然不是我們希望發生的事情，通常而言一個「正常」的遞迴不會深得很誇張才是，因此大多數時候系統會對 stack 區塊設定一個大小限制，如果一個程序使用的 stack 記憶體量超過該限制就強制把該程序結束並且回傳錯誤信號，這便是 stack overflow 的由來了。

在一些演算法中，我們有時候會遞迴得誇張地深 (通常發生在對一張圖 DFS)，高達數百萬層之類的。對一般的系統而言這個深度當然很可疑，因此許多系統中會強制把這樣子的程序結束掉，形成 Runtime Error 的誤判。這種時候常見的處理方法有幾種：

1. 如果本身就是系統管理者，可以透過修改設定解決此一問題。
2. 如果本身並不具有管理員權限，但是是特定 OS (如 Windows) 且只需要繳交可執行檔，那麼可以透過修改編譯器設定解決此一問題 (P.S. Windows 中 stack 的記憶體限制是記錄在可執行檔裡的，可以在編譯時期加設定修改)。
3. 如果以上兩者皆非，那麼有三種方式：
   (a) 使用特殊指令改變 stack 使用的記憶體，如：組合語言。
   (b) 換成一個不需要極深遞迴的演算法 (例如 DFS 換成 BFS，如果可行的話)。
   (c) 自己實做 stack，模擬遞迴。

如果對組合語言不熟悉，又是在競賽期間或者是於線上評測系統獲得 Runtime Error 的成績，最常使用的還是 3.(b)、3.(c)，也因此有一兩次自己實做遞迴的經驗是很重要的。

2 虛擬記憶體

2.1 動機

如果我們進一步地思考作業系統會怎麼實做上述的記憶體布局，我們很快就會發現一個問題　到底 heap 的底部和 stack 的頂部要相隔多遠才好呢？假如記憶體從 0 編號到 99，每一個程式 heap 底下最少要 10 單位記憶體，stack 以上最少要 15 單位記憶體，而現在有一個程式 heap 的底部設定在 10，stack 的頂部設計在 49。那麼實際上這個程式只要使用 heap 和 stack 兩個部份的記憶體總量超過 40 單位，整個程式就會因為把 heap 和 stack

的記憶體用罄而產生記憶體錯誤或者不足的問題。明明記憶體有 100 單位，怎麼只能使用 $10 + 15 + 40 = 65$ 單位的記憶體呢？這樣子如果只有要跑這個程式，剩下的 35 單位記憶體就形成了徹徹底底的浪費，花了錢買記憶體，結果只買到了一場心碎的騙局。

如果想要避免這樣的問題，直接把兩者之間的距離拉到盡量遠，將 heap 的底部設定在 10，stack 的頂部設定在 84，原先的問題就徹底解決了。然而，新的問題是，如果同時這時有第二個程式要執行，我們該如何配置這個程式的記憶體位置呢？不管怎麼配置，都一定會插在第一個程式的某些記憶體區間內，這樣不是造成第一個程式記憶體錯誤，就是讓 heap 和 stack 的大小再次受到了實質限制 (總不能寫到第二個程式的記憶體還繼續寫吧？要知道，通常你的電腦都只有一塊記憶體，也就是說所有程式都要共用它)。

從上面兩段的例子我們就可以知道，記憶體布局實際上是相當困難的問題：我們或者必須要自斷手腳，限制每個程序的記憶體使用量，好讓作業系統能輕易地配置空間，或者就得要犧牲對多程序的支援，再不然就得要放棄記憶體連續配置的記憶體布局，讓程式變得極度難以設計。這一切的萬惡來源都在於每個程序都直接操縱了實體的記憶體，兩個程式之間彼此互相無法溝通，卻要爭奪有限的資源，自然會產生諸多摩擦、激情與火花。

2.2 解決方法

仔細想想就會發現，整個問題的癥結事實上在於我們要求「連續區間」的便利性。如果概念上拿到的記憶體不是連續的區間，我們撰寫程式時就很難知道還有哪些記憶體可以用，想要夠紮實地使用記憶體也變得困難。試想你跟 OS 要了一塊長度 100 的陣列，結果陣列不是連續的，第一格在 a[0]，第二格在 a[5]，第三格在 a[100] 之類的，會非常麻煩。

但就像情侶間的關係一樣，如果 A 和 B 是情侶，B 雖然實際上腳踏兩條船但卻能做到讓 A 渾然不覺，那麼到底 B 的腳踏兩條船又有什麼關係呢？對於取用資源的一方而言，只要它所要求的條件和資源都能夠達成，真相是什麼真的有那麼重要嗎？

這便是作業系統做的魔法了。作業系統一樣在每個程式執行時會依據程式指定的記憶體位址對記憶體進行操作，但實際上真正被操作的記憶體位址是作業系統自行決定的，它只要做到讓程式看起來事情有被順利執行即可。舉例而言，某個程式可能會以為它自己可以操作的記憶體位址是 [0, 100] 這個區間，但是當它要求操作 0 這個位址時，可能實際上被操作的位址是 2217；當它要求操作 1 這個位址時，可能實際上被操作的位址是 514；當它要求操作 2 這個位址時，可能實際上被操作的位址是 689..... 只要作業系統能夠保證兩個對程式而言不一樣的記憶體位址永遠都不會對應到一樣的實體記憶體位址，且兩個對程式而言一樣的記憶體位址永遠都對應到一樣的實體記憶體位址，實際上真正被操作的記憶體位址為何對程式而言根本就不重要了！而程式看到的記憶體位址並非實體位址的特色，讓我們對應於實體記憶體稱呼這樣的記憶體為「虛擬記憶體 (virtual memory)」。

2.3 更有彈性的記憶體

在原本的記憶體配置中，為了所有的可用記憶體區間都是連續的，作業系統必須一口氣配置好連續的記憶體區塊，否則事後配置的記憶體就會和一開始的記憶體區間不連續；在虛

擬記憶體系統下，記憶體只是「看起來」是連續的，實際上是離散的，便可以更加彈性地配置了，只要作業系統好好地維護虛擬與實體位址的對照表即可。

舉例而言，一樣記憶體有 100 單位大小，現有兩個程式 A, B，各自先用了基本的 $10 + 15$ 單位記憶體 (承續上面的假設)，那實體記憶體中可能作業系統會先分別配置那基本的 50 單位在 [0, 49]。接著 A 可能呼叫了某個函式，需要用 2 單位的記憶體，作業系統可以把這 2 單位配置在實體的 [50, 51]；如果接下來 B 動態配置了 10 單位的記憶體 (malloc, new 之類的語法)，作業系統可以把這 10 單位配置在實體的 [52, 61]。對作業系統而言，所有的程式的各種記憶體都交錯在一起，並沒有太大的分別，它只須記得「某個程式口中的某個記憶體位址其實是某某位址」就可以了。

從上面的例子我們可以看到，記憶體其實到了真正需要的時候再進行配置即可，因此一開始作業系統就可以先「表面上」讓每個程式都有跟總記憶體大小一樣大的空間，但卻不需要真的幫每個程式都預留這麼大的空間。

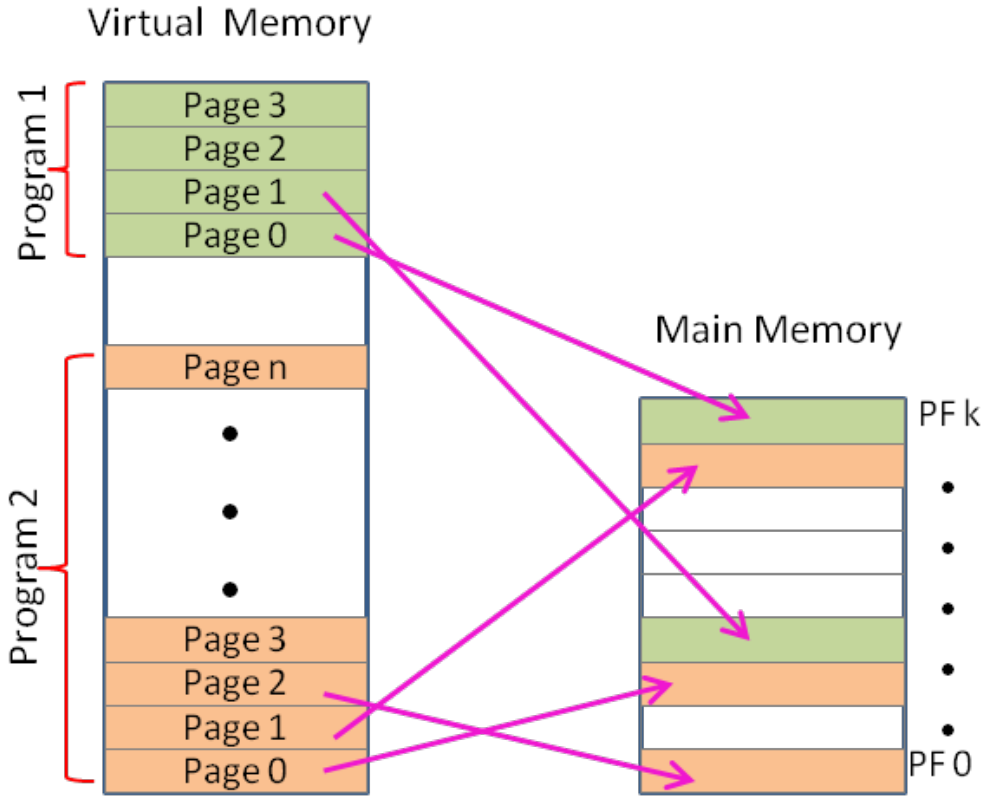


Figure 2: 可以看到程式在使用著連續的記憶體，但實際上這些記憶體在實體記憶體中是離散的。而兩個程式也可以用同個虛擬地址，但卻對應到不同地方。

2.4 共享記憶體

一個很常見的情景是，兩個程序會共用同一份資料。這種情況下，如果我們在兩個程序中都把這些資料載入記憶體，同一份資料不就佔了兩份空間嗎？有了虛擬記憶體的概念，我

們只需要把這兩個程序裡面資料儲存的位址通通都指向同一塊實體記憶體，就可以在兩方都維持獨立性的情況下避免掉許多不必要的浪費了！(※ 想一想：如果兩個程序互相不知道對方的存在，也不清楚記憶體可能會跟別人共用，會有什麼樣的危險？怎樣的記憶體才能夠安心地共用呢？)

你有想過兩個獨立的程式是如何溝通的嗎？透過共享記憶體，我們也能讓兩個程式進行溝通！(就像是一個留言板一樣，一個人寫字，另一個人就能看到他寫了什麼，甚至還能擦掉後寫新的東西。)

2.5 更大的記憶體

在原先的模型中，我們存取的都是同一塊記憶體，一般而言實體上對應的是 RAM (Random Access Memory)。RAM 雖然相當快速，但壞處是空間相當不充足，很容易有用完的一天。記憶體如果用不夠了，程式當然跑不下去，難道就放任程式毀滅嗎？虛擬記憶體再次挺身而出：**跨越不同的儲存裝置**

與其讓程式崩壞，我們不如退而求其次使用比較慢但是容量更大的儲存裝置，如硬碟。傳統的 RAM 要超過 128 GB 已經是非常昂貴的了，相較之下硬碟 2 TB 的價格卻便宜非常多。如果我們可以把這麼巨大的容量拿來使用，當然彈性會大增，然而使用原先直接操作實體記憶體的方法，我們很難做到「跨越不同的記憶體裝置」這樣的事情。

有了虛擬記憶體，我們可以在作業系統這端進行管理，超出實體記憶體負荷的部份我們就將其指向其他不同儲存裝置，這樣便能夠在不影響程式的抽象性的情況下（也就是說，寫程式時仍然在對記憶體做操作），讓程式實質存取不同的記憶硬體。

進一步地，使用者通常很少會一口氣積極地使用很多個程式，多數的程式在多數的時間都是處於停滯、等待或者低效能運算的狀態。也因此一個常見的技術是，在 RAM 不足時，我們總是把當前最頻繁存取的幾個程式放在 RAM 裡，其餘的程式放在硬碟裡，並且在程式狀態改變時改變儲存的位置 (例如原先 A 的多數區塊在 RAM，B 的多數區塊在硬碟，但接下來要 B 要大量存取記憶體，所以把 B 的多數區塊搬到記憶體來，A 的多數區塊搬到硬碟去，以加快當前執行的 B 速度)。如果是直接存取實體記憶體的架構，此時實體位址會有極大量的改變，A 和 B 都必須要大規模地重整記憶體配置；但在虛擬記憶體的框架下，對 A 和 B 這兩個程序而言，除了存取資料時不知道為什麼速度產生了改變，整個世界看起來並無分別，大大減輕了開發者的負擔。

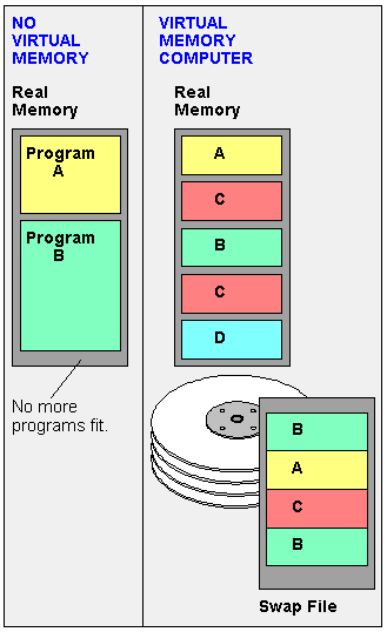


Figure 3: 一個簡單的有無虛擬記憶體的比較圖

習題

1. 閱讀完記憶體布局的方式後，請推論並回答下列問題（只需回答選項，不需回答原因）：

   (a) (10 pts) 下列敘述何者為真？

   (A) 記憶體布局存在標準，所有作業系統都必須遵循標準的布局系統

   (B) 在同一個作業系統上且使用相同方式編譯後，不同的 C 語言程式可能會有不同排列的記憶體布局

   (C) 有時候遇到遞迴過深的問題可以使用修改程式以外的方式解決

   (D) 如果遞迴太深，會把 heap 的記憶體用光，導致 runtime error

   (b) (10 pts) 關於各個記憶體區塊的特性，下列何者為非？

   (A) 一樣的 code 在不同編譯器下編譯後形成的執行檔大小不一定一樣，因為不同編譯器轉換成機器碼的方式可能不同

   (B) 靜態變數和區域變數如果沒有指定初始值，預設有可能會是任意的值

   (C) 因為編程者指定的初始值會存在 initialized data segment，所以手動初始化的變數越多，一般而言編譯後的執行檔會越大

   (D) 在程式開始執行後，位於 initialized data segment 和 uninitialized data segment 的資料都還是有可能會被修改

   (c) (10 pts) 下列基於上述記憶體布局的推論，何者正確？

   (A) 如果我們同時運行兩個相同的程式 (但不一定做相同的操作，如同時開兩個 skype 視窗跟不同的人聊天)，那麼我們可以透過兩者共享相同的 initialized segment data 記憶體來節省空間

   (B) 如果我們同時運行兩個相同的程式 (但不一定做相同的操作，如同時開兩個 skype 視窗跟不同的人聊天)，那麼我們可以透過兩者共享相同的 text segment 記憶體來節省空間

   (C) 假如某個程式中有兩個 function $main(), f()$，並且執行順序為：

   1. 執行 $main()$
   2. 呼叫並執行 $f()$
   3. 退出 $f()$，回到 $main()$
   4. 退出 $main()$，程式結束

   那麼在執行期間，任意 $f()$ 內的變數的記憶體位置一定都比任意 $main()$ 內的變數的記憶體位置還要小

   (D) 在 Linux 上我們可以透過修改編譯參數來增加 stack 深度的上限

2. 有份 C 程式碼如下：

```c
int var1;
static int var2;
int var3 = 3;

int f(int var4){
    if( var4 <= 1 ) return 1;
    else return f(var4-1) + f(var4-2);
}

int main(){
    int var5;
    static int var6;
    int *var7 = (int*)malloc(sizeof(int)*100);
    int var8 = 5;
    return 0;
}
```

假設編譯器沒有做任何變數的優化，請問程式碼中，各個變數分別儲存於記憶體分佈中的哪個區塊呢？請使用代號 (A~E) 回答問題。

選項：

(A) Text Segment

(B) Initialized Data Segment

(C) Uninitialized Data Segment

(D) Heap

(E) Stack

(a) (5 pts) var1

(b) (5 pts) var2

(c) (5 pts) var3

(d) (5 pts) var4

(e) (5 pts) var5

(f) (5 pts) var6

(g) (5 pts) var7

(h) (5 pts) var8

(i) (5 pts) *var7

(j) (5 pts) main

(k) (5 pts) main()