PROJECT_3

周易廷

Part A – 位元運算

1. 操作說明:

   甲、在 terminal 輸入 gcc page.c -o page 進行編譯

   乙、輸入./page –n 16 即可在 group23_ans.txt 看到結果，如果不打數字，預設為 16。

2. 程式概念:

   此程式根據使用者輸入的參數 n，將 test.txt 中的 address 分成兩部分，n 為頁偏移量的 bit 數，剩下的 bit 數為頁數的 bit 數。

- Example (n = 16)

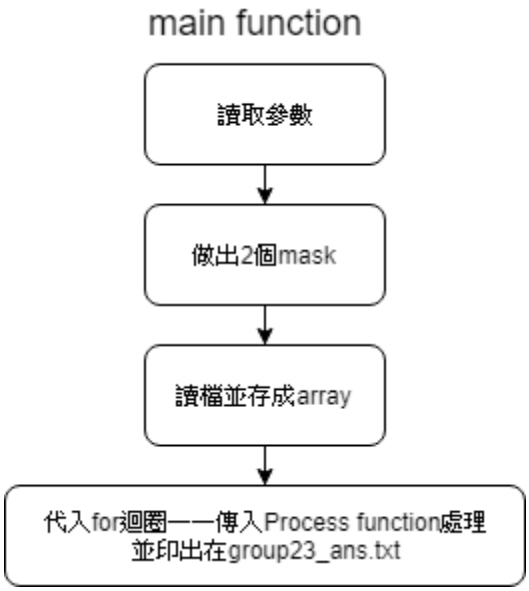
Address (32bit)	0x7a30cffa➔0111 1010 0011 0000 1100 1111 1111 1010
頁數/頁偏移量 (2 進位)	0111 1010 0011 0000 / 1100 1111 1111 1010
頁數/頁偏移量 (10 進位)	31280 / 53242

首先在處理 test.txt 中地址的部分，先利用 fscanf 將 address 讀取並存成字串，再利用 union 搭配 int 與 char array 資料型別將字串轉成實際的值，先根據 ASCII 碼將字元轉成 16 進位的對應值(Example: ‘a’ – ‘W’ = 10)，一次對兩個字元作處理(∵char 是 8 bit, int 是 32 bit, 16 進位是 4bit)，將較大位元的字元往左位移 4 次，再將兩者 or 起來存入對應的 char array index，此步驟經過 4 次迭代即可將字串轉成正確數值。

在將 address 分成頁數與偏移量的部分，先使用 bitwise operator << 將 1 左位移再跟 1or 起來，經過 n 次即可做出長度為 n 的全 1 的值，再使用 operator ~ ，即可得到剩餘部分皆為 1 的值。

有了上述的兩個值，就可以開始對 address 做 mask 的動作(將 address 與上面兩個值 and 起來)，即可完成。

3. 程式流程圖



4. 執行結果:

```
 1 0X00000000          0           0
 2 0Xffffffff          1           2147483647
 3 0Xffff0000          1           2147418112
 4 0X0000ffff          0           65535
 5 0X00ffff00          0           16776960
 6 0X7a30cffa          0           2050019322
 7 0Xdf9fb945          1           1604303173
 8 0X7838db03          0           2016992003
 9 0X67201ac3          0           1730157251
10 0Xd976f0e3          1           1500967139
11 0X7e793d82          0           2121874818
12 0X4c384a8e          0           1278757518
13 0X52474f58          0           1380405080
```

Part B – Cache Simulator

1. 操作說明:

   甲、將 csim.c 放入 cachelab，在 terminal 輸入 make 完成編譯，輸入./test-csim 觀看結果。

2. 程式概念:

   此程式先定義兩種 structure 儲存輸入資料:

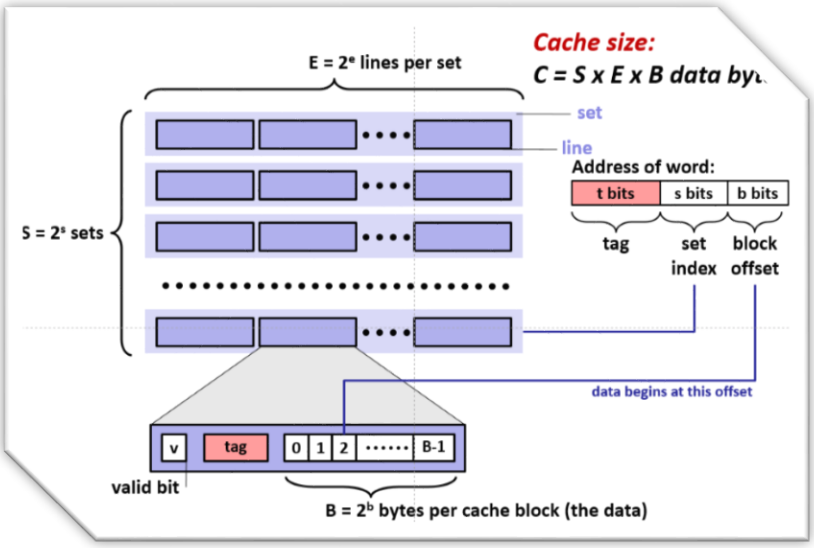
   I. CacheInfo: 儲存輸入參數
   II. Input: 儲存 Instruction、address、size，方便傳入函式。

   再定義三種 structure 模擬 cache 結構:

   I. Block: 儲存 Block 指標、tag
   II. Set: 儲存 Block 數量、指向 Block list 中第一個的指標。
   III. Cache: 儲存指向 Set 陣列的指標。

   讀取 trace file 結果的部分，利用 fscanf 讀取%llu 類型，將 Instruction、address、size 存入 Input 結構，其中 address 為 64bit 的 unsigned long long int。

   根據參考資料 https://hackmd.io/@drwQtdGASN2n-vt_4poKnw/H1U6NgK3Z?type=view，cache 結構如下:



因此在處理輸入的 trace 結果，先根據輸入的參數使用 bitwise operator，將

tag、set id 的部分截取出來:

I. Address 往右位移(s + b)次可得 tag

II. Address 往左位移(64 – (s+b))次，再往右位移(64 - s)次可得 set index

- Example (s = 4, E = 1, b = 4)

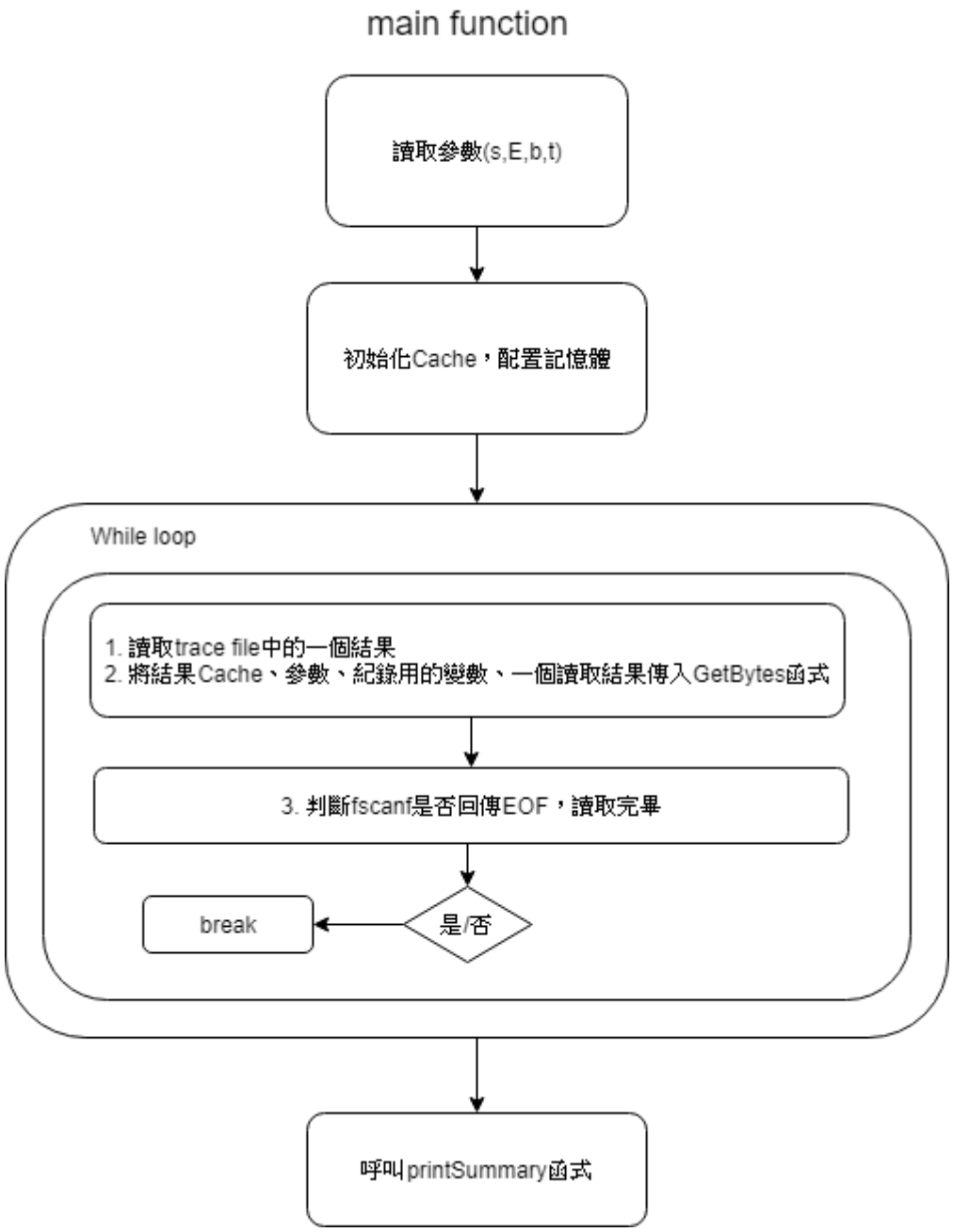
Address(16 進位)	7fefe0594
Address(2 進位)	0111 1111 1110 1111 1110 0000 0101 1001 0100 Tag set id block offset
Tag(往右位移(s + b)次)	0000 0000 0111 1111 1110 1111 1110 0000 0101
Set index(往左位移(64 – (s+b))次，再往右位移(64 - s)次)	0000 0000 0000 0000 0000 0000 0000 0000 1001

完成擷取 set index 與 tag 之後，即可到對應的 set 中尋找是否有對應的 tag，若有，則此次存取為 Hit，若沒有則為 Miss，其中 Miss 情況又分為 2 種，set 中 line 數達到上限則須根據 LRU 選擇犧牲掉的 line，若還未達到上限，則直接將 memory 中的資料搬進 set 中的 block 再根據 block offset 選擇要存取的 bytes，然而由於此程式只是模擬 cache 行為，因此僅需將 tag 存入 block 即可。

在實現 LRU 的部分，使用 link list 的方式，儲存 block，可節省大量記憶體空間(不須事先配好每個 set 中的 block 空間)，也可同時實現 LRU。在選擇哪個 block 要做為 victim 時，僅須根據 link list 中順序決定(類似 queue)，將 link list 中第一個 block 作為犧牲者，而每次新進來的 block 則放入尾端，若一個 block 發生了一次 Hit，也將此 block 放入尾端，如此一來，距離上次使用最久的 block 將永遠在 link list 的第一個。

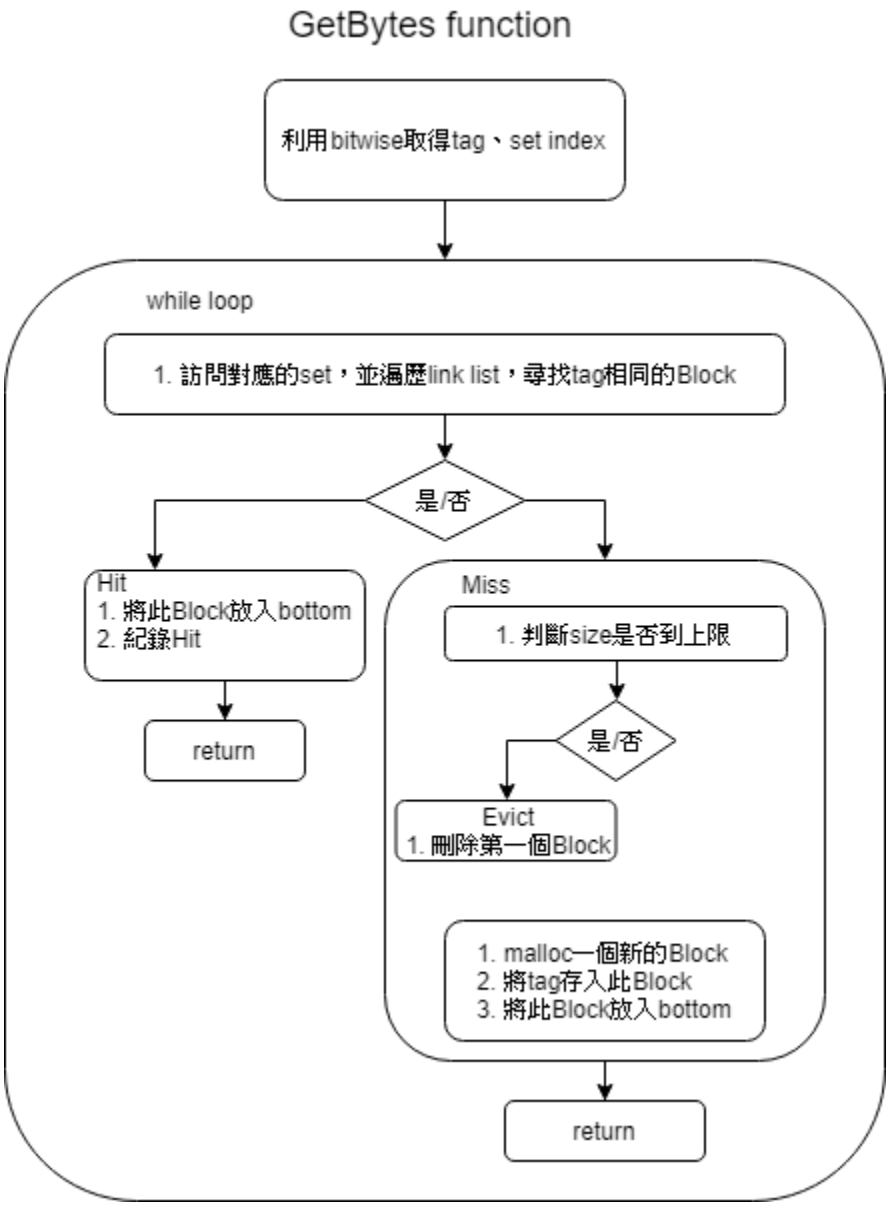
最後根據紀錄 Hit, Miss, Evict 的部分，將變數以傳參考的方式送入模擬的 funtion 紀錄；其中需特別注意，當 Instruction 為 Modifiy 時，可視為一次 Load 加上一次 Store，因此在 Load 的部分若為 Hit，在 Store 部分也為 Hit，一共 2 次，若在 Load 部分發生 Miss 的情況，在 Store 的部分時，cache 已經將資料抓近來，所以必為 Hit 的情況，一共一次。

3. 程式流程圖
   I. main function:



II.

GetBytes function:



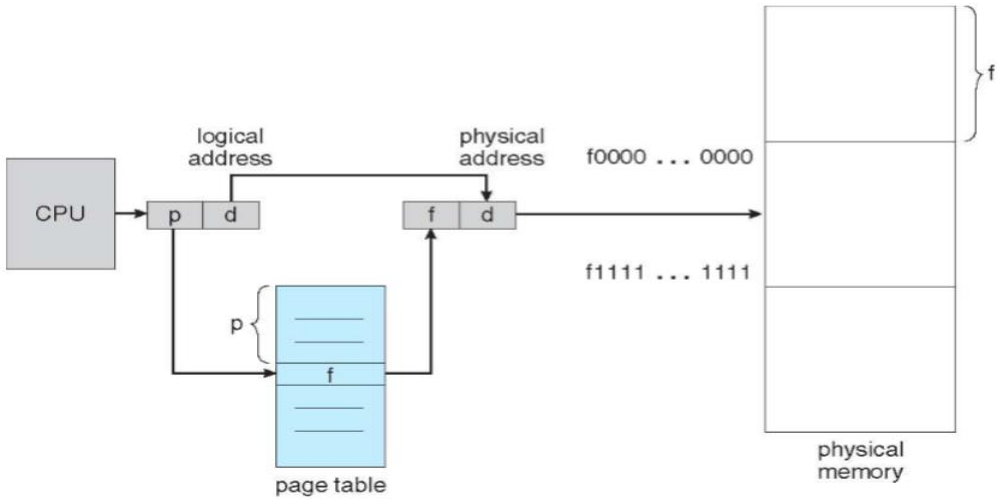
4. 執行結果

```
e24056051@e24056051-VirtualBox:~/cachelab$ ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
     6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    27

TEST_CSIM_RESULTS=27
e24056051@e24056051-VirtualBox:~/cachelab$
```

Part C – Huge Page

OS 會將 logical memory 中的資料分成好幾塊(blocks)稱為 pages，同理，在實體記憶體中也將資料分成好幾個 frames。Pages 與 frames 是互相對應的，因此大小也相同。當 OS 需要用到資料時，OS 必須將之載入 memory 中，反之不需要時則將資料從 memory 移至 disk。作業系統會設置 page table，轉換 logical address 變成 physical address。page table 會以兩個參數 page number 跟 page offset 找出 page 對應的 frame 在哪裡。



由於 process 的 address space 都是存在 virtual 中，當 process 要被執行時 OS 或 CPU 就需要將它轉換到 physical 中，因此 OS 需要知道每個 process 對應的 page 還有該 page 存放在哪裡。就如同我們到圖書館想借書時，要先找一下館藏的目錄(virtual)，從目錄中得知該書在圖書館的實際位置在哪一區哪一個書架上

(physical)，而當今天有很多的目錄時，就必須花更多的時間才能找到你要的書；同理，如果今天有很多的 pages 時，OS 要搜尋需要的 page 就會花很多時間。傳統的 page 大小為 4069 bytes，所以 1MB 的記憶體含有 256 個 pages、1GB 的記憶體含有 256000 個 pages，假設今天有個 process 使用 1GB 的記憶體，而每個 page table entry 消耗 8bytes，那麼 OS 在找 page 時就必須消耗約 2MB。

有兩種方式可以提高 OS 在處理較大的資料:1.增加 disk 中 page table 的 entries 2.增加 page size。由於目前的硬體限制，processor 最多僅能支援數千個 page table entries，所以第一個方法較不易執行。而第二個方法即是所謂的 Huge pages(在 Windows 系統又稱 Large pages)。Huge pages 的概念就是把每個 pages 的大小變大，根據 kernel 版本或者硬體架構的不同，可以從 2MB~256MB 都有。同樣以借書來比喻，如果今天借書目錄的每一頁 Size 都變大，那麼每頁可以記錄的書就變多，在書本的總數目不變的情況下，目錄就會變小本，不用翻很多頁，就可以更快找到你要的書。

Huge pages 與 page 主要的差別:

1. Page table 的大小: Huge page < page
2. TLB miss: Huge page < page 由於 Huge page 的 table 較大，因此資料所佔的內存大小也變大，提高 TLB 的命中。

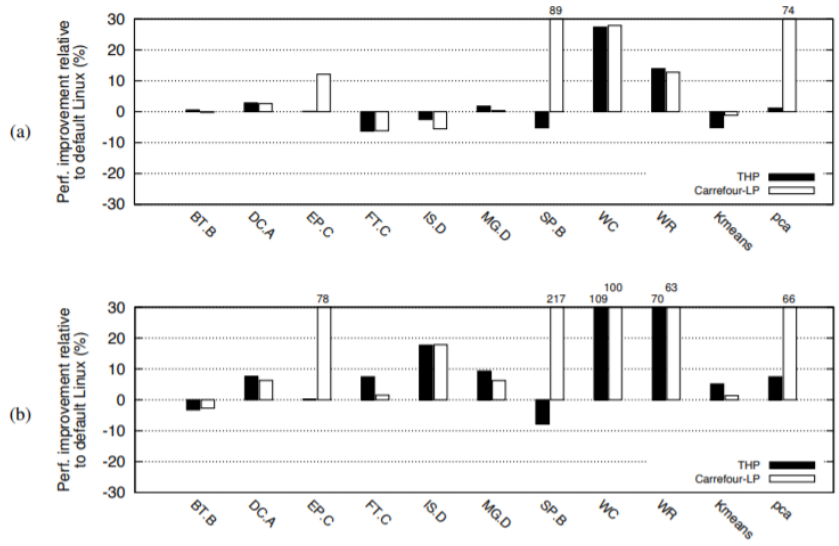
- Fewer TLB misses

Page size	512 entries coverage	1024 entries coverage
4KB (default)	2MB	4MB
2MB	1GB	2GB
1GB	512GB	1024GB

3. Huge pages 不會被 swap，所以可以減少 OS 的消耗。

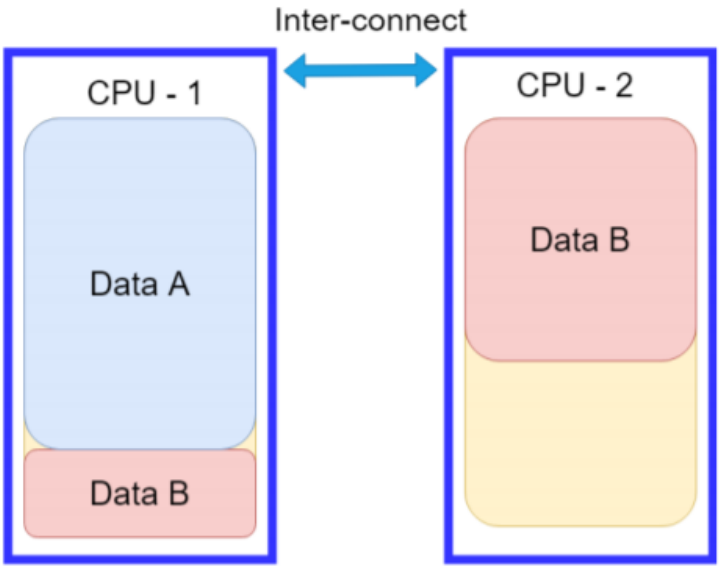
Huge pages 的缺點:

1. 在某些應用環境下會降低效能，尤其是在較頻繁操作的應用環境中，由於 CPU 可能回爭搶同一個 Page，造成 CPU 的效率降低



2. Poor locality:

   由於 Huge page 的 page size 增加，可以存放的資料也增加，因此假設有一筆資料在原本 4k 的 page 中剛好占滿一個連續的區域，而改存在 Huge page 中時因為每一頁的空間變大，因此可能有一部分的資料會被存到上一頁的剩餘空間中，造成不連續的狀態，同筆資料放在兩個不同的 Pages，這樣如果要讀取完整的資料時，就必須透過 Inter connect 去獲得每個不同部分的資料，使系統效能降低。



Huge pages 使用案例:

在運作 Oracle 時，常常都會利用 Huge pages 來解決伺服器記憶體過大的問題，由於 Linux 是以分頁存取來管理記憶體，CPU 會以 LRU 演算法來將不常

用的記憶體換到虛擬記憶體中，而經常用到的資料則保存在記憶體中，達到物盡其用的目的，讓記憶體的充分使用效率可以提升。但是如果記憶體過大時，分頁就會很多，影響 CPU 效率，因此會透過 Huge pages 來解決。