

---

---

---

---

---



1. A process can be swapped as long as it's not in the running state.

T 2. system call 都有從 user mode 到 kernel mode 的過程沒錯。當 user 有需要做與 OS 有關的事，就要先轉到 kernel mode 才可以執行。

F 3. Variable size memory allocation will cause internal fragmentation.

F 4. API calls 都是 system call.

有些 API call 是 call 到不涉及 system call 的 (math function)

5. TLB 翻 full virtual to full physical memory addr.

T b. both page fault and segmentation fault are handled by OS.

page fault 是找不到有效的 physical memory

segmentation fault 是取到不屬於自己記憶體空間的 fault

兩者都是由 OS 管控，runtime 由 OS 檢查到才會報錯

F 7. The memory address sent out from CPU is ~~physical address~~, ~~virtual address~~.

透過 MMU, CPU 送出的 virtual address 才能轉成 physical address，對記憶體內容存取。

F 8. POSIX interface OS 可以有不同的 system calls.

POSIX interface 的 system call 要一樣，才可使 interface

基於此 interface 寫的程式能跨平台執行。

(但實作方法可以不一樣)

2. (a) 因為程式執行通常會有兩個部份，一是計算(使用CPU).

另一則是 I/O(不使用CPU)，當一程式做 I/O 時，在 multi programming 的架構下，另一程式可以去使用 CPU，提升 CPU 使用效率。

(b) 程式如果大多在 memory 裡面，每個程式分配到的記憶體空間會很少，導致執行時不斷發生 Page Fault，整個系統的多數程式只能不斷做 I/O，沒辦法使用到 CPU。  
即

### 3. (a) 如何造成

#### 1. 多元化的功能

kernel 只留下最少部份的功能，意味著使用者可更弹性定义自己的功能

#### 2. 有限的計算資源

(kernel fail 的機率低)

#### 3. 放在有惡意使用者之處運行

kernel 留下極少的功能，安全性增加。

### (b) 為何要

犧牲效能造成他的長度

因為此系統設計目的是scalable、易維護、可根據不同

需求引入不同功能至 user 端。

4. 什麼是 Privileged instruction? kernel mode 才能做的事情

- ✓ (a) If
- ✓ (b) Turn off all the interrupt
- ✗ (c) Generate any trap instruction → 像 open 就用到了?  
(不确定)
- ✓ (d) Context Switch
- ✓ (e) Write memory contents
- ✗ (f) Read status of processor

(不确定)

5. Message Passing 比 shared memory 來得慢的原因?

1. 涉及 system call, 傳送到 OS, 及後收來自 OS 的訊息

2. 需要做 memory copy

b.

1. 因為 thread 需要 copy 的資料比 process 少 (少了 code 和 data segment)

2. one-to-one 比較好。

因為 I/O 代表要叫 Thread 到 Waiting State

採用 many-to-one 的架構, 當遇到 I/O event,

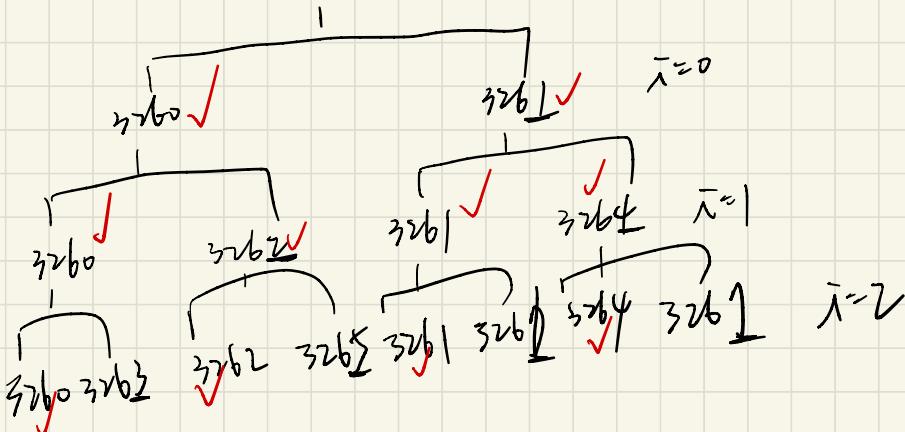
kernel 會 sleep, CPU 就 idle 了, 效率低。

i.e. CPU 使用

D.

return 0 是 child process  
 return pid 時是 parent process  
 (child的)

main: 3260



## E. Page Fault

- ① 因為發生 page fault 要從 disc copy 到 memory  
 非常耗時 (比起從 memory 到 Register 來說)  
 時
- ② 當一個 page 的 dirty bit 为 1，且要被換掉，我們不需將該  
 page 的內容更新到 secondary storage 上，節省了 page  
 fault 所要花的時間

③ 為什麼 page fault 低？

因為程式行為特性。一般來說，程式在一段時間內會重覆使用到某些特定的資料，所以程式想拿資料

時，大部份時候資料已經因為前面的 page fault 被送進 memory 或 TLB 了。

④ 大一點的 page size 之優劣？

缺點：internal fragmentation 更容易發生，浪費記憶體空間，page fault rate ↑

優點：page table 會比較小，儲存 page table 的記憶體空間所佔用

會降低。有更多儲存空間，page fault rate 應會下降

9.

(a) 一段時間內被 access 到的 pages 的組合

(b) 當所有程式的 working set 總合大於剩餘記憶體空間時，OS 就會將一些程式從記憶體移開，釋出空間給其它還在跑的程式使用，就比較不會有 thrashing 現象發生。

(c) working set size too large, problem?

代表 multi-programming degree 降低，  
CPU 使用率會下降。

10. (hierarchical, hashed, inverted)

(a) embedded system on single application

空間資源有限

inverted page table 最好。

因為 hierarchical 會因為 outer page table 浪費空間。  
多一個

hashing 也會 linked list 佔用一些空間

(b) A general purpose computer system with huge memory size

hashing 好。

因為 inverted 和 hierarchical page table 都要花上很多時間  
才能找到 entry。而且 inverted 不能 share read-only  
的 data, 會多存很多一樣的 data。

11.

page size = 400 bytes

pure demanding paging and LRU (Least Recently Used)

(1) 4 frames X, 1, 2, 3

↓ ↓ ↓  
code A[0] ~ A[299]

① bring in code fault

② page 1: A[0] ~ A[99] = 1 fault

③ page 2: A[100] ~ A[199] = 1 fault

④ page 3: A[200] ~ A[299] = 1 fault

⑤ page 4: A[300] ~ A[399] = 1 fault

⑥ page 2: A[100] ~ A[199] = 1 fault

⑦ A[200] ~ A[299] = 0 (page 3) no fault

⑧ A[300] ~ A[399] = 0 (page 4) no fault

⑨ A[100] ~ A[199] = 0 (page 2) fault (page 2) 2

⑩ A[0] ~ A[99] = 0 (page 1) fault (page 1)

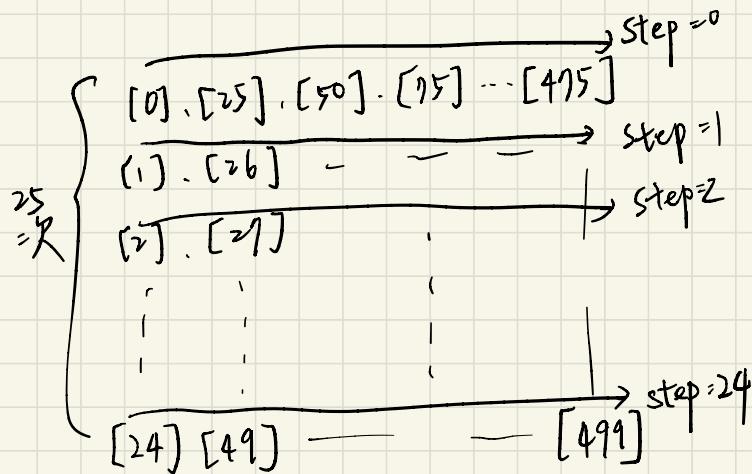
= 8 faults

500 個 整 數  
(1個 page 可容

約  $\frac{400}{4} = 100$  個  
整數

6

(2)



load code fault

Step = 0

~~[0] [25] [50] [75] [100] [125] [150] [175]~~

~~fault~~      ~~fault~~

(cod) 1 fault  
for 1 (step)  $25 + 5 = 125$  faults

for loop (step)  $25 \times 5 = 125$

"

~~[200] - - - [225] [250] [275]~~

~~fault~~      ~~fault~~

~~[400] - - - [425] [450]~~

~~fault~~

25 faults

Step = 1

~~[1] [26] [51] [76]~~

~~fault~~

- - - fault

- - - - fault

⋮  
Step = 24

5 faults

(b) 6 frames

① code 1 frame (fault)

for loop ① 5 frame available

Step = 0

page 1 [0]. [25]. [50]. [75] fault

page 2 [100] - - - [175] fault

page 3 : | fault

page 4 : | fault

page 5 [400] - - - [475] fault

Step = 1 ~ 24

由於 A 已全放在 page 5 里，故

不再有 page fault

for loop 2 Step 0 ~ 24 同上理由, # of page fault = 0

⇒ 6 page fault.

12.

logical (12 bit): 

00	01	0000	1010
----	----	------	------

physical (10 bit)

Segment number: 0 → (a)

seg[0] = base: 00 1100 0000  
limit: 10 0000 0000

256Bytes page size  
→ 8bit

page table entry: 3

limit > offset → true

→ valid access

offset + base = 01 0000 1010  
00 1100 0000

(b) 

01	1100	1010
----	------	------

 → linear address

page[1] = 3 ↓

(c)

(d) 

11	1100	1010
----	------	------

 → physical address

(e) since linear address length = 10

page size =  $2^8$  Byte (8 bit / page)

→ 2 bits for page number  $\rightarrow 2^2 = 4$  entries

(f) ex. 

seg num	seg offset
00 11 1111 1111	

→ seg[0] = base: 00 1100 0000  
limit: 10 0000 0000

since limit < offset, this is an invalid logical address.  
(out of bound error)

