

# 期末考

## Pseudo code (CH7)      Bounded Buffer Problem

(a) producer - consumer

### Producer

While (true) {

  ...

  /\* produce an item in next\_produced \*/

  ...

  wait (empty); 初值 n

  wait (mutex); 初值 1

  ...

  /\* add next\_produced to the buffer \*/

  ...

  Signal (mutex);

  signal (full); 初值 0

}

## Consumer

```
While (true) {  
    wait (full);  
    wait (mutex);  
    ...  
    /* remove an item from buffer to next_consumed */  
    ...  
    Signal (mutex);  
    signal (empty);  
    ...  
    /* consume the item in next_consumed */  
    ...  
}
```

吃飯想字幕

# Readers - Writers Problem

## Writer

```
while (true) {  
    wait (rw_mutex);  
    ...  
    /* writing is performed */  
    ...  
    signal (rw_mutex);  
}
```



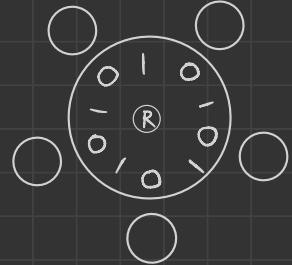
## Reader

```
while (true) {  
    Wait (mutex);  
    read_count++;  
    if (read_count == 1) /* first reader */  
        Wait (rw_mutex);  
    signal (mutex);  
    ...  
    /* reading is performed */  
    ...  
    Wait (mutex);  
    read_count--;  
    if (read_count == 0) /* last reader */  
        signal (rw_mutex)  
    signal (mutex);  
}
```

## Dining - Philosophers Problem 哲學家就餐問題

The structure of Philosopher i :

```
while (true) {  
    Wait (chopstick [i] ) ;  
    Wait (chopstick [(i+1)% 5] ) ;  
  
    signal (chopstick [i] ) ;  
    signal (chopstick [(i+1)% 5] ) ;  
}
```



- 他們的生活全部用於思考和吃飯
- 他們不和彼此互動
- 若某人感覺飢餓時，就試圖使用最靠近他的筷子
- 每次只能拿取一支筷子

# CH8

## Banker's Algorithm 銀行家演算法



$$* \text{need} = \text{max} - \text{allocation}$$

				(3,3,2)						
	Allocation			Max	Available	Need	順序			
	A	B	C	A	B	C	A	B	C	
T <sub>0</sub>	0	1	0	7	5	3	10	5	7	7 4 3 5
T <sub>1</sub>	2	0	0	3	2	2	5	3	2	1 2 2 1
T <sub>2</sub>	3	0	2	9	0	2	10	4	5	6 0 0 4
T <sub>3</sub>	2	1	1	2	2	2	7	4	3	0 1 1 2
T <sub>4</sub>	0	0	2	4	3	3	10	4	7	4 3 1 3

1. Available (3,3,2) 先給 T<sub>1</sub> Need (1,2,2), 下次就有 (3,3,2)

$$+ T_1 \text{ Allocation } (2,0,0) = (5,3,2)$$

$$2. (5,3,2) \text{ 給 } T_3 \Rightarrow (5,3,2) + (2,1,1) = (7,4,3)$$

$$3. (7,4,3) \text{ 給 } T_4 \Rightarrow (7,4,3) + (0,0,2) = (7,4,5)$$

$$4. (7,4,5) \text{ 給 } T_2 \Rightarrow (7,4,5) + (3,0,2) = (10,4,7)$$

$$5. (10,4,7) \text{ 給 } T_0 \Rightarrow (10,4,7) + (0,1,0) = (10,5,7)$$

∴ 序列  $\langle T_1, T_3, T_4, T_2, T_0 \rangle$  是安全準則

T<sub>1</sub> Request (1,0,2) : 先檢查 request 是否小於 available ( $(1,0,2) \leq (3,3,2)$ )

	Allocation			Need			Available			順序
	A	B	C	A	B	C	A	B	C	
T <sub>0</sub>	0	1	0	7	4	3	7	5	5	4
T <sub>1</sub>	$\underbrace{(2,0,0)}_{(2,0,0)+(1,0,2)}$	0	$\underbrace{2}_{(1,2,2)-(1,0,2)}$	0	$\underbrace{2}_{(1,2,2)-(1,0,2)}$	0	5	3	2	1
T <sub>2</sub>	3	0	2	6	0	0	10	5	7	5
T <sub>3</sub>	2	1	1	0	1	1	7	4	3	2
T <sub>4</sub>	0	0	2	4	3	1	7	4	5	3

1. Available (2,3,0) 先給 T<sub>1</sub> Need (0,2,0), 下次就有 (2,3,0)

$$+ T_1 \text{ Allocation } (3,0,2) = (5,3,2)$$

$$2. (5,3,2) \text{ 紿 } T_3 = (5,3,2) + (2,1,1) = (7,4,3)$$

$$3. (7,4,3) \text{ 紿 } T_4 = (7,4,3) + (0,0,2) = (7,4,5)$$

$$4. (7,4,5) \text{ 紿 } T_0 = (7,4,5) + (0,1,0) = (7,5,5)$$

$$5. (7,5,5) \text{ 紿 } T_2 = (7,5,5) + (3,0,2) = (10,5,7)$$

$\therefore \langle T_1, T_3, T_4, T_0, T_2 \rangle$  是 safe sequence

\* Can request for (3,0,0) by T<sub>4</sub>? 不夠借

\* Can request for (0,2,0) by T<sub>0</sub>? 乘 (2,1,0), 沒法再借給其他人

$\Rightarrow$  unsafe

# CH 8



## Deadlocks 死結

系統中存在一組 process 陷入互相等待對方所擁有的資源的情況，造成所有的 process 無法往下執行，使得 CPU 利用度大幅降低。

\* 滿足以下 4 個條件：

1. Mutual exclusion 互斥

一個資源一次只能被一個 process 所使用

2. Hold and wait 占用與等候

取得一個資源之後等待其他的資源

3. No preemption 不可搶先

資源只能由 process 自己釋放，不能由其他方式釋放

4. Circular wait 循環式等候

每個 process 都握有另一個 process 請求的資源，導致每一個 process 都在等待另一個 process 釋放資源

# ✓ CH9 Paging

Page numbers

offset

frame

number

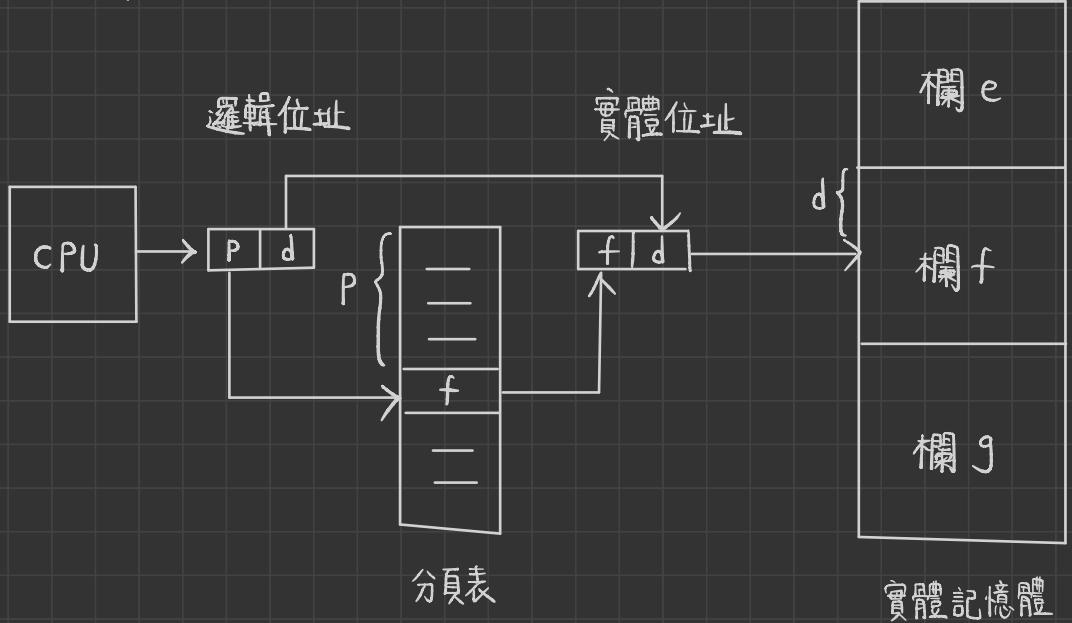
(a) 何謂 Paging ? 畫出 hardware ?

Paging : 分頁是一種記憶體管理方法，可以使電腦

的主記憶體可以使用儲存在輔助記憶體中的資料，解決外部斷裂跟處理各種大小的行程

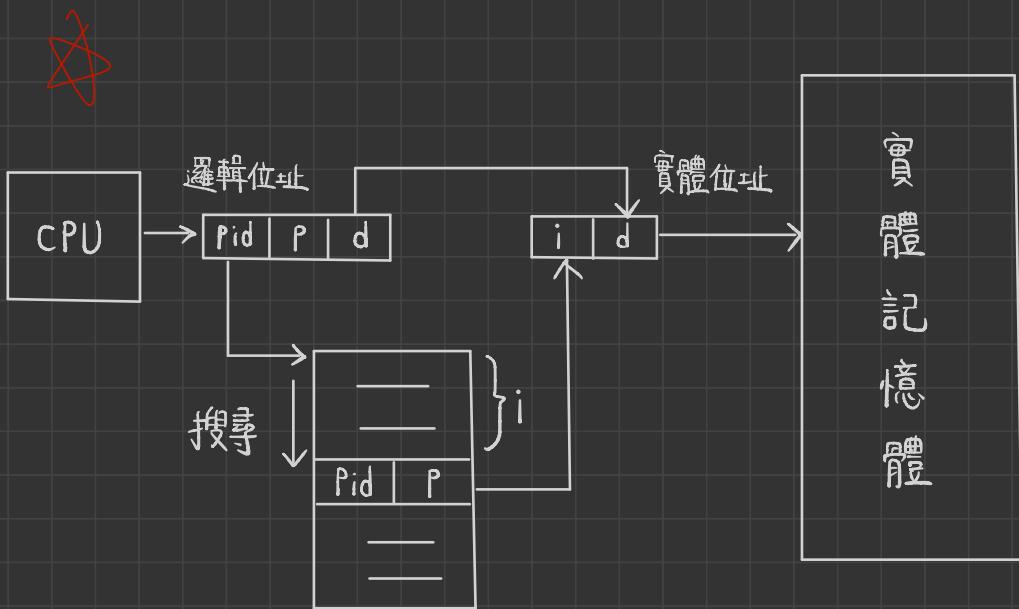
the NC transform

## ▲ 分頁硬體



## ✓ (b) inverted page table 反轉分頁表

是以 physical address 為對象，建立一個 page table 給所有 process (有 m 個 frame, 就有 m 個 table entry)，每個 entry 都會紀錄著 process id 跟 page number



## Dynamic storage-allocation problem 動態儲存體配置問題

指如何從可用區間去滿足n個大小

- first-fit 最先配合

配置第一個夠大的區間。搜尋的起始位置可為第一個可用區間或是前一個最先配合搜尋動作的結束位置，而找到夠大的區間後，立即停止搜尋的動作

- best-fit 最佳配合

在所有容量夠大的區間中，將最小的一個區間分配給行程。

最佳配合策略會找出最小的剩餘區間

- worst-fit 最差配合

將最大的區間配合給行程。

最差配合策略會找出最大的剩餘空間，而這些區間的用處可能比最佳配合法找到的剩餘區間來得大

9.6

將記憶體劃分：為六個區塊，分別為300KB、600KB、350KB、200KB、750KB和125KB（按順序），最適配合、最佳配合、最差配合、的演算法將如何放置大小為115KB、500KB、358KB、200KB和375KB的行程（按順序）

First-fit

115KB 放入 300KB, 剩下 185KB, 600KB, 350KB, 200KB,  
750KB, 125KB

500KB 放入 600KB, 剩下 185KB, 100KB, 350KB, 200KB.  
750KB, 125KB

358KB 放入 750KB, 剩下 185KB, 100KB, 350KB, 200KB  
392KB, 125KB

200KB 放入 350KB, 剩下 185KB, 100KB, 150KB, 200KB  
392KB, 125KB

375KB 放入 392KB, 剩下 185KB, 100KB, 150KB, 200KB  
17KB, 125KB

將記憶體劃分：為六個區塊，分別為300KB、600KB、350KB、  
200KB、750KB和125KB（按順序），最適配合、最佳配合、最差配合、  
的演算法將如何放置大小為115KB、500KB、358KB、200KB和375KB  
的行程（按順序）

Best fit

115KB 放入 125KB，剩下 300、600、350、200、750、10

500KB 放入 600KB，剩下 300、100、350、200、750、10

358KB 放入 750KB，剩下 300、100、350、200、392、10

200KB 放入 200KB，剩下 300、100、350、0、392、10

375KB 放入 392KB，剩下 300、100、350、0、17、10

將記憶體劃分：為六個區塊，分別為300KB、600KB、350KB、200KB、750KB和125KB（按順序），最適配合、最佳配合、最差配合、的演算法將如何放置大小為115KB、500KB、358KB、200KB和375KB的行程（按順序）

Worst fit

115 放入 750，剩下 300、600、350、200、635、125

500 放入 635，剩下 300、600、350、200、135、125

358 放入 600，剩下 300、242、350、200、135、125

200 放入 300，剩下 300、242、150、200、135、125

375 必須 wait

# CH 9

- Logical address 邏輯位址

CPU 產生的位址

- Physical address 實體位址

記憶體單元看到的位址

CH9

## Fragmentation 斷裂

外部斷裂是指記憶體空間不連續，而無法配置給行程；

內部斷裂是指被額外配置給行程，但實際並未被使用的記憶體空間。兩者相同的地方在於，都有不會被使用，而形同浪費的記憶體空間。

CH10

## Virtual memory 虛擬記憶體

是電腦系統記憶體管理的一種技術。它使得應用程式認為它擁有連續的可用的記憶體（一個連續完整的位址空間），而實際上，它通常是被分隔成多個實體記憶體碎片，還有部分暫時儲存在外部磁碟記憶體上，在需要時進行資料交換。

✓ CH10

## Page replacement 分頁替換

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1  
(page)

- FIFO 先進先出演算法：先進來的優先替換掉

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	4	4	4	0	0	0	1	1	1	1	7	7	0	0

- Optimal page-replacement algorithm 最佳分頁替換演算法  
看未來會不會用到，最久沒用的優先換掉

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	4	2	0	2	0	0	1	2	7	0	1	1	0

- Least recently used (LRU) algorithm 近來最少使用演算法  
換掉最遠的 (optimal相反)

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	0	4	4	4	0	3	3	3	1	1	0	1	0	0	1

## ✓ CH 10

### Thrashing 輾轉現象

(a) Thrashing 發生的時機？

當各行程的總需求欄位空間超過實體記憶體的欄位空間時，便會發生輾轉現象。

當一個行程花在分頁的時間比執行時間更高

(b). Working-set model 如何處理 thrashing ?  
(工作集模型)

大約評估各個行程在不同的執行時期需要多少的欄位，並提供足夠的欄位防止問題產生。

使用一個參數  $\Delta$ ，來定義工作集欄框，決定工作集的精確度，而其最重要的性質就是其大小。

工作集合是用來預測該行程接下來要使用的需求欄位數量(demand frames)，作業系統(OS)依照各個行程的工作集合來分配他們的欄位數量，以避免輾轉現象發生。

Thrashing：會發生在 page 分配到 frame 不夠多時，會很常發生 page fault

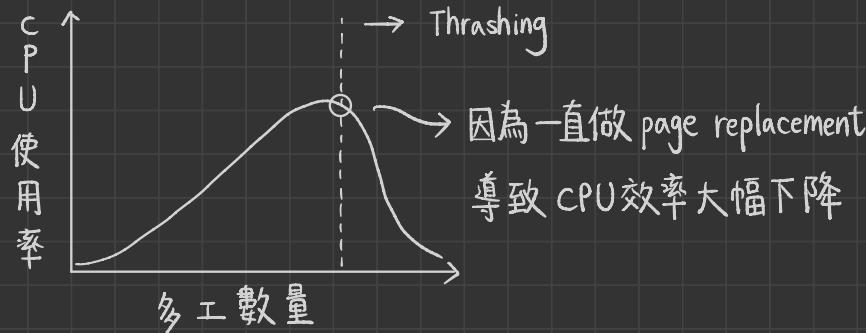
① Frame 不夠，發生 page fault



② Page replacement 發生

③ 很快 frame 又不夠，又 replacement (一直重覆)

④ 導致 CPU 使用率 ↓



# CH16

加密演算法組成元件：

- 鑰匙的集合 K
- 訊息的集合 M
- 密文的集合 C
- 加密函數 E: K → (M → C)
- 解密函數 D: K → (C → M)

## ✓ Encryption 加密

### • Symmetric encryption algorithm 對稱加密演算法

傳送方與接收方的加解密皆使用同一把密鑰

### • Asymmetric encryption algorithm 非對稱加密演算法

每個使用者擁有一對金鑰 - 公鑰和私鑰，訊息由其中一把金鑰加密後，必須由另一把金鑰解密

### • Authentication 認證演算法

限制訊息可能傳送者的集合就稱為認證。

對於證明訊息尚未被修改是很有用的。

### • Digital signature algorithm 數位簽章演算法

對訊息進行簽名（使用私鑰加密），驗證者用公鑰解密。

如果和原消息一致，則驗證簽名成功

### • Message digest (hash value) 訊息摘要（雜湊值）

是一個唯一對應一個訊息或文本的固定長度的值。

它由一個單向雜湊加密函式對訊息進行作用而產生

## ✓ RSA Algorithm :

1. 選擇兩個大質數  $P, Q$
2. 計算  $N = P \times Q$
3. 選擇公鑰 (public key)  $E$ , 使它不為  $(P-1)$  或  $(Q-1)$  的因數
4. 選擇私鑰 (private key)  $D$ , 讓右邊算式成立  $(D * E) \% [(P-1) * (Q-1)] = 1$
5. 加密過程為：密文  $(CT) = \text{明文} (PT)^E \% N$
6. 解密過程為：明文  $(PT) = \text{密文} (CT^D \% N)$

$\text{mod}$  「同餘」

$$Ex. 4 \div 3 = 1 \cdots 1$$

$$10 \div 3 = 3 \cdots 1$$

$$4 \equiv 10 \pmod{3}$$

$$5 \div 2 = 2 \cdots 1$$

$$7 \div 2 = 3 \cdots 1$$

$$5 \equiv 7 \pmod{2}$$

## RSA 演算法步驟

Ex. 1. 選二個質數  $P=7, Q=13$

$$2. N = P \times Q = 91 \quad \text{and} \quad (P-1)(Q-1) = 72$$

3. 選一個比  $72$  小的質數做為 public key ( $k_e$ ),

此數必須與  $72$  互質, 選 5

4. 再選 private key ( $k_d$ ), 使  $k_e \cdot k_d \pmod{72} = 1 \Rightarrow 29$

$$k_e, k_d, N = 5, 29, 91$$

5. message 加密金鑰的次方  $\div N = \text{加密數} (m^{k_e/k_d} \div N) ... \frac{A}{B}$

## Security violation categories

- Breach of confidentiality 機密的缺口

違反類型包括未授權資料的讀取

- Breach of integrity 完整的缺口

牽涉違反未授權資料的修正

- Breach of availability 可用的缺口

牽涉違反未授權資料的破壞

- Theft of service 服務的竊盜

牽涉違反未授權資料的使用

- Denial of service 拒絕服務

牽涉防止合法使用該系統

- Bounded - Buffer Problem 垃圾桶理論

  - n buffers, each can hold one item

  - 3 個 semaphore, 預設 mutex = 1, full = 0, empty = n

將資料放進去

do {

/\* produce an item \*/

wait(empty);

wait(mutex);

/\* add next produced \*/

signal(mutex);

signal(full); } while(true);

將資料取出來

do {

wait(full);

wait(mutex);

/\* remove an item \*/

signal(mutex);  
signal(empty);

/\* consume an item \*/ } while(true);

- Readers and Writers Problem

檔案可以接受同一時間被多程序讀取，

但讀跟寫只能擇一

① Readers: 只能讀，不能寫。

② Writers: 能讀寫，但不能同時

- 2 個 semaphore, Semaphore rw\_mutex = 1, mutex = 1  
reader count = 0

- rw\_mutex: 控制 reader / writer 進入

- mutex: 控制 reader 修改 reader count, 不被其他 reader 影響

```

// reader
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex); // 其他 reader)
    /* reading */
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while(true)

// writer
do {
    wait(wr_mutex);
    /* writing */
    signal(wr_mutex);
} while(true)

```

## △ Starvation

First variation: 如果有很多 reader 一直排，那 writer 將無法 write (因為 wr\_mutex)

Second variation: 只要 writer 準備就緒，立刻 write。  
前面的 reader 無法 read.

- Dining Philosophers Problem

5個哲學家，5個筷子，每個哲學家只思考或吃

- Bowl of rice (data set)
- Semaphore chopsticks [5] initialized to 1

```
while(true){
```

```
    wait(chopsticks[i]);
```

```
    wait(chopstick[(i+1)%5]);
```

```
    signal(chopstick[i]);
```

```
    signal(chopstick[(i+1)%5]);
```

⇒ 會出現僵局、互斥問題

- Monitor DiningPhilosophers {

```
enum {Thinking, Hungry, Eating} state[5];
```

```
condition self[5];
```

```
void pickup (int i){
```

```
    state[i]=Hungry;
```

```
    test(i);
```

```
    if(state[i]==Eating) self[i].wait;
```

```
void putdown (int i){
```

```
    state[i]=Thinking;
```

```
    test((i+4)%5);
```

```
    test((i+1)%5); } (test 左右鄰居)
```

```
void test (int i){ if(state[i]==Hungry)&&
```

```
(state[(i+4)%5]==Eating)&&(state[(i+1)%5]==Eating)
```

```
{ state[i]=Eating;
```

```
self[i].signal(); } }
```

Hungry 且左右都非 Eating

```
initialization_code{ }
```

→ 开始 Eat

```
for (int i=0; i<5; i++)
```

```
    state[i]=Thinking;
```

```
}
```

Dining Philosophers. pickup( $i$ );

/\* Eat \*/

Dining Philosophers. putdown( $i$ );

$\Rightarrow$  No deadlock, 但 starvation is possible

總結：

- ① 初始化：哲學家在跟彼此客氣，都在發呆
- ② 有人餓了：test自己是不是真的餓了，然後看左右兩邊  
    是不是已經動筷子了，沒有就 Eat
- ③ 吃飽了：放下筷子然後看他們要不要吃

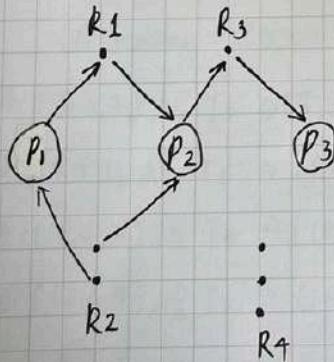
## Deadlocks

\* 滿足以下 4 個條件：

1. Mutual exclusion：一個 resource 一次只能被一個 process 用
2. Hold and wait：process 至少取到一個 resource，然後等其他 process 擁住的 resource
3. No Preemption：釋放資源時，是 process 自願的，沒有被中斷。
4. Circular wait：每個 process 都在等另一個 process 擁住的 resource 釋放 ( $P_1$  等  $P_2$ ,  $P_2$  等  $P_3$  等)，這也代表 single process 不會有 deadlock

## Resource-Allocation Graph (RAG)

- ① request edge：process 指向 resource
- ② assignment edge：resource 指向 process



\* No cycle  $\rightarrow$  No deadlock

Cycle  $\rightarrow$  ① Resource 只有一個 instance : deadlock

② Resource 多個 instance : 有可能 deadlock

## Deadlock Prevention

- Mutual Exclusion: 不共享資源
- Hold and Wait: Process 必須保證在要求資源時，不能佔用其他資源，除可以一次取得所有資源
- No Preemption: 只要 process 掌握有一些資源，但得不到其他的，就要放棄全部，再重新申請
- Circular Wait: 對 resource type 強制安排線性順序，不讓 circular wait 條件達成

## Banker's Algorithm:

- available [1...m]: 表示 resource type 還有幾個 Instance 可以使用
  - max [1...n, 1...m]: 表示 process 需要最多幾個 resource type 的 Instance
  - allocation [1...n, 1...m]: 表示 process 現在擁有幾個 resource type 的 Instance
  - need [1...n, 1...m]: 表示 process 現有還要多少 resource type 的 Instance
- \* need = max - allocation

	Allocation			Max			Available(3,3,2)			Need			順序
	A	B	C	A	B	C	A	B	C	A	B	C	
T <sub>0</sub>	0	1	0	7	5	3	10	5	7	7	4	3	5
T <sub>1</sub>	2	0	0	3	2	2	5	3	2	1	2	2	1.
T <sub>2</sub>	3	0	2	9	0	2	10	4	5	6	0	0	3
T <sub>3</sub>	2	1	1	2	2	2	7	4	3	0	1	1	2.
T <sub>4</sub>	0	0	2	4	3	3	10	4	7	4	3	1	4.

① Available(3,3,2) 先給 T<sub>1</sub> (Need 1,2,2), 下次就有 (3,3,2) + T<sub>1</sub>

$$(Allocation \ 2,0,0) = (5,3,2)$$

② (5,3,2) 給 T<sub>3</sub>  $\Rightarrow (5,3,2) + (2,1,1) = (7,4,3)$

③ (7,4,3) 給 T<sub>2</sub>  $\Rightarrow (7,4,3) + (3,0,2) = (10,4,5)$

④ (10,4,5) 給 T<sub>4</sub>  $\Rightarrow (10,4,5) + (0,0,2) = (10,4,7)$

⑤ (10,4,7) 給 T<sub>0</sub>  $\Rightarrow (10,4,7) + (0,1,0) = (10,5,7)$

$\therefore < T_1, T_3, T_2, T_4, T_0 >$  是 safe sequence

T1 Request (1, 0, 2) : 先檢查 Request 是否小於 Available  $((1, 0, 2) \leq (3, 3, 2))$

Allocation	Need			Available $(2, 3, 0)$			順序
	A	B	C	A	B	C	
T0	0	1	0	7	4	3	7 5 5
T1	3	0	2	0	2	0	5 3 2
	$(2, 0, 0) + (1, 0, 2)$						1
T2	3	0	2	6	0	0	10 5 7
T3	2	1	1	0	1	1	7 4 3
T4	0	0	2	4	3	1	7 4 5

$\langle T_1, T_3, T_4, T_0, T_2 \rangle$  是 safe sequence

\* Can request for (3, 3, 0) by T4? 不夠借

\* Can request for (0, 2, 0) by T0? 剩 (2, 1, 0), 沒法再借給其他人  
 $\Rightarrow$  unsafe

How to satisfy a request of size  $n$  from list of free holes?

• First-fit: Allocate the first hole that is big enough

• Best-fit: Allocate the smallest hole that is big enough;

must search entire list, unless ordered by 0

• Worst-fit: Allocate the largest hole; must also search entire list

\* First-fit 跟 Best-fit 就速度與儲存率來說都比 worst-fit 好

① 分配足夠大的第一個 hole ② 分配足夠大的最小 hole, 必須搜尋整個列表, 除非是按尺寸搜尋 ③ 分配最大的 hole, 也需要搜尋整個列表

### 兩種 Fragmentation 解釋 (空間浪費)

• External Fragmentation: 存在整個記憶體空間中去滿足請求，但不連續。

• Internal Fragmentation: 被分配的記憶体比請求的記憶体要大一些，而這個尺寸差異是分區內部的記憶体，但並未使用。

### Address Space

• Logical address: 由 CPU 產生, virtual address 虛擬位置

• Physical address: 真正記憶体位置, 由 memory unit 所看見

Deadlock: 競爭資源時造成一些 process 不能動

Mutual exclusion: 一個 resource 只能被一個 process 用

Hold and wait: process 至少取到一個 resource, 然後等其他 process 擁住的 resource

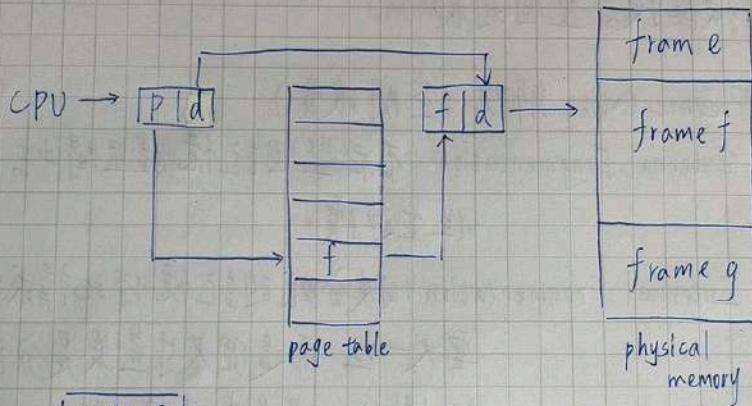
No Preemption: 釋放資源時，是 process 自原真的，沒有被中止。

Circular wait: 每個 process 都在等另一個 process 握住的 resource  
釋放

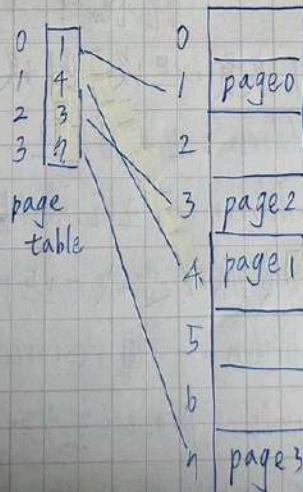
Paging: 是為了解決 fragmentation (external, 但沒有 internal) 跟處理各種大小的 process

- Frames: 將實体 (physical memory) 切成大小固定的 block
- Pages: 將邏輯 (logical memory) 切成大小相同的 block

Frames 跟 Pages 大小相同，當 program 需  $n$  pages  $\Rightarrow$   $n$  個 free frames



page 0
page 1
page 2
page 3
logical memory



優點：

1. 不用照 懶
2. 不用連續
3. X 外部 waste

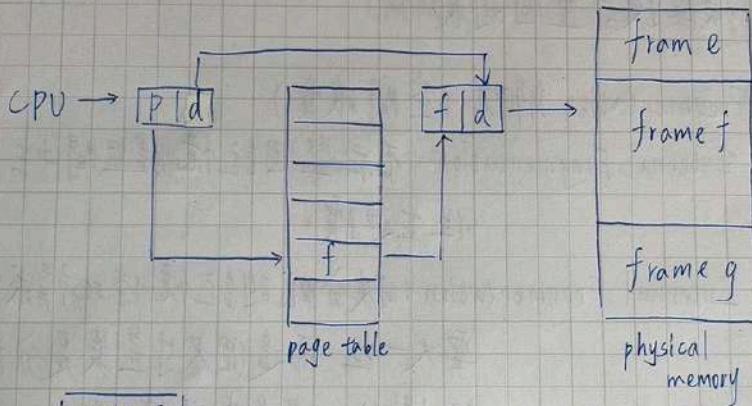
No Preemption: 釋放資源時，是 process 自原真的，沒有被中止。

Circular wait: 每個 process 都在等另一個 process 握住的 resource  
釋放

Paging: 是為了解決 fragmentation (external, 但沒有 internal) 跟處理各種大小的 process

- Frames: 將實体 (physical memory) 切成大小固定的 block
- Pages: 將邏輯 (logical memory) 切成大小相同的 block

Frames 跟 Pages 大小相同，當 program 需 n pages  $\Rightarrow$  n 個 free frames



page 0
page 1
page 2
page 3
logical memory

0	1	0
1	4	1
2	3	2
3	7	3
		page0
		page1
		page2
		page3

page table

優點：

1. 不用照 懶
2. 不用連續
3. X外部 waste