面試整理

1

```
#define MUX(a,b) a*b
求MUX(10+5,10-5)=?



10+5*10-5 = 10+50-5 = 55
```

2

```
int fun(int x){
   int count = 0 ;
   while(x){
      count++ ;
      x = x & (x-1) ;
   }
   return count ;
}
求fun(456)+fun(123)+fun(789)=?



456 = 111001000
455 = 111000111
123 = 1111011
122 = 1111010
789 = 1100010101
788 = 1100010100
=> 每次減少一個1
=> 4 + 6 + 5 = 15
```

3

```
int c;

int fib(int n){
    c++;
    if ((n==1)|| (n==2))
        return 1;
    return (fib(n-1)+fib(n-2));
}

int main(){
    c = 0;
    fib(5);
    printf("%d", c);
    return 0;
}
```

```
fib(1) => c+1
fib(2) => c+1
fib(3) => c+1, fib(2), fib(1) => c+3
fib(4) => c+1, fib(3), fib(2) => c +1 +3 +1 => c+5
fib(5) => c+1, fib(4), fib(3) => c +1 +5 +3 => c+9
```

4

什麼是OS?

確保 Process 可以正確執行，不會讓 Process 跟 Process 之間互相干擾，
並透過 kernel mode 跟 user mode 保護硬體，
並提供 high level 的 system call 讓使用者不能直接操作硬體，
簡化操作，也更加有效率等。

4

Program, Process and Thread?

Process 是 OS 分配 resource 的單位，相對的 Thread 是 OS 分配 CPU-time 的單位。
Process 之間的溝通相對複雜，要嘛是跟 OS 要一塊 Shared Memory，
不然就是透過 OS Message passing，
而 Thread 之間的溝通相對簡單，只要透過 Global Variable 即可，
雖然可能會有些問題（Race Condition）不過整體是比較簡單的，
再者 Thread 的切換可能不用轉到 Kernel Mode（看 Thread 如何實作）
又 Process 切換需要儲存許多資料到 PCB 而 Thread 相對少，
所以 Thread 的 Context Switch 也比 Process 快。

Program：放在二次儲存裝置中，尚沒有被Load到記憶體的一堆Code
稱之為「程式」。 (也就是還是死的)

Process：已經被Load到記憶體中，任何一行Code隨時會被CPU執行，且其宣告的在記憶體的變數的值會隨著需求而不斷變動。
稱之為「程序」。 (也就是活的Program) => 恐龍本第三章
一個多工作業系統(Multitasking Operating System)可以同時運行多個Process
然而一個CPU一次只能做一件事情，但CPU的數量永遠少於運行中的Process數，
因此每個Process使用的時間需要被排程(Scheduling) => 恐龍本第五章
又每個Process間在記憶體中，如果擺放的方式不當，就會在記憶體中產生很多沒辦法用到的碎片，因此MemoryManagement是一個問題 => 恐龍本第八章
另外，每個Process所需要的記憶體總合，也可能大於實體記憶體，因此需要另外用二次儲存裝置充當虛擬記憶體(Virtual Memory)，但是二次儲存裝置的速度肯定很慢，因此如何做到對虛擬記憶體最小的依賴，盡量避免Page Fault(電腦在主記憶體中找不到資料，而要去二次記憶體找，就稱為Page Fault)
防止Thrashing的發生(因為Virtual Memory演算法不當，造成幾乎每次存取都要依賴二次記憶體，就是Thrashing)，以達到效能最佳化，也是個學問 => 第九章

Thread ：在同一個Process底下，有許多自己的分身，就是Thread，中文又翻成執行緒。
以往一個Process一次只能做一件事情，因此要一面輸入文字，一面計算字數，這種事情是不可能的。但是有了Thread之後，可以在同一個Process底下，讓輸入文字是一個Thread，計算文字又是另外一個Thread，對CPU來說兩個都是類似一個Process，因此兩個可以同時做。
又一個Process底下有數個Thread，而一個Process的Global Variable可以讓它的所有Thread共享，也就是所有Thread都可以存取同一個Process的Global Variable。而每個Thread自己也有自己的專屬Variable。 => 恐龍本第四章
但是，如果有兩個Thread要存取同一個Global Variable，有可能發生問題，也就是說可能會存取到錯的值(例如兩個Thread同時要對一個Variable做加減，最後那個答案可能會是錯的)，這就是Synchronization問題 =>恐龍本第六章
又，每一個Thread之間可能會互搶資源，而造成死結(Deadlock)，只要以下四個條件都滿足就有死結。(1)這個資源不能同時給兩個人用 (2)有一個人拿了一個資源，又想拿別人的資源 (3)如果一個人占了茅坑不拉屎，占用資源很久，仍不能趕他走 (4)A等B，B等C，C等D，D又等A 等成一圈。 要解決這種狀況有Avoid(預防) 或 避免(Prevent)兩種方式，破除以上四種其中一種即可。
=> 恐龍本第七章

```
int main(int argc , char* argv[]){
        pid_t pid;
        printf("Process fork!!\n");
        pid = fork();

        int signal;
        if(pid<0){
                printf("Fork Failed!");
                return 1;
        }
        else if(pid==0){
                printf("Child process executes Test program!!\n");
                printf("Hello I'm parent process , my pid = %d\n\n",getppid());
                execvp(argv[1],argv);
        }
        else{
                wait(&signal);
                printf("Receving the SIHCHLD signal\n\n");
                if(WIFEXITED(signal)!=0){//Normal Exit
                        printf("Normal termination with exit status = %d\n\n",WEXITSTATUS(si
                }
                else{//Abnormal Exit
                        check_signal(WTERMSIG(signal));
                }
        }

        return 0;
}
```

4

哲學家就餐問題

死結 :哲學家從來不交談，這就很危險，可能產生死結，
每個哲學家都拿著左手的餐叉，永遠都在等右邊的餐叉（或者相反）

活鎖 :如果五位哲學家在完全相同的時刻進入餐廳，並同時拿起左邊的餐叉，
那麼這些哲學家就會等待五分鐘，同時放下手中的餐叉，再等五分鐘，又同時拿起這些餐叉。
缺乏餐叉可以類比為缺乏共享資源。一種常用的電腦技術是資源加鎖，
用來保證在某個時刻，資源只能被一個程式或一段代碼存取。
當一個程式想要使用的資源已經被另一個程式鎖定，它就等待資源解鎖
當多個程式涉及到加鎖的資源時，在某些情況下就有可能發生死結。
例如，某個程式需要存取兩個檔案，當兩個這樣的程式各鎖了一個檔案，
那它們都在等待對方解鎖另一個檔案。

服務生解法
一個簡單的解法是引入一個餐廳服務生，哲學家必須經過他的允許才能拿起餐叉。
因為服務生知道哪只餐叉正在使用，所以他能夠作出判斷避免死結。

資源分級解法
在哲學家就餐問題中，資源（餐叉）按照某種規則編號為1至5，
每一個工作單元（哲學家）總是先拿起左右兩邊編號較低的餐叉，
再拿編號較高的。用完餐叉後，他總是先放下編號較高的餐叉，再放下編號較低的

Chandy/Misra解法
把筷子湊成對，讓要吃的人先吃，沒筷子的人得到一張換筷子券。
餓了，把換筷子券交給有筷子的人，有筷子的人吃飽了會把筷子交給給券的人。
有了券的人不會再得到第二張券。保證有筷子的都有得吃。

5

講解一下如何避免 Race Condition

解決此問題的基本概念，便是讓共享的資源在不同執行緒內可以受到控制，
但有時因為設計不良，便會出現競爭危害（race hazard）。又稱為競爭條件（race condition）
OS本身有提供 Semaphore 跟 Monitor 只要使用得當就可以避免這樣的問題。

下列程式會產生 race codition問題，我們無法確定最後 var 的值是 10 或是 20?

```
Parent thread:                          Child thread:

int var; // global variable

// create child thread
pthread_create(...)
var = 20;                               var = 10;
                                        exit
pthread_join(...)
printf("%d\n", var);
```

直覺的解法便是讓兩段設定var的程序有先後關係，如下例，
透過訊息的傳遞會讓設定 var 值的程序有 "happens-before" 關係，
也就是 var = 20; 一定會在 var = 10; 之前發生。因此最後的結果固定會是10。

```
Parent thread:                          Child thread:

int var;

// create child thread
pthread_create(...)
var = 20;
// send message to child
                                        // wait for message to arrive
                                        var = 10;
                                        exit

// wait for child
pthread_join(...)
printf("%d\n", var);
```

所以，若是某個記憶體內的資料，會同時被兩個不同的 thread進行存取，
我們可以先檢查這兩個 thread 寫入同份資料時是否存在 "happens-before relation"，
若不存在此關係，便存在 race condition.

6

講解一下什麼是 (Pipeline) Hazard

分三種，Structural hazards, Data hazards, Control hazards。
Structural hazards
硬體資源不夠多而導致同一時間內要執行的多個指令無法執行，是先天限制。
如記憶體一次就只能給一個人讀取。

Data hazards
會發生在 lw 後面直接接 add 或是 branch 等，
在資料就緒之前就要使用，就會出現 Data hazards。
在連續數個指令都存取同一個暫存器時可能會發生的 Race Condition。
會造成 執行時 read/write 指令的順序與在組語之中的順序不同。
Data Hazard 又分為 RAW, WAW, WAR 三種，其中以 RAW 最為常見。
i1. R2 <- R1 + R3
i2. R4 <- R2 + R3
(R2 可能還沒寫入完全)
Hardware approach：forwarding，不用等到前一個instruction執行到最後一個stage，
在ALU算出result後就將它送入下個instruction中當作input。
Software approach：rearrange。

Control hazards
當我們需要某個指令的結果來作一些決定，可是這個指令還在執行，無法馬上提供所需要的結果。
發生在會修改 Program Counter 的指令（Jump/Branch/else）
(a)stall：如果知道是一個branch instruction，則暫停直到正確的condition知道後才開始下一個in
(b)predict：事先預測branch instruction不會發生，如果預測結果錯誤則放棄已經進入pipeline的指
(c)delayed branch：在branch instruction之後先執行下個合適的指令，但是必須確保這個指令是安全

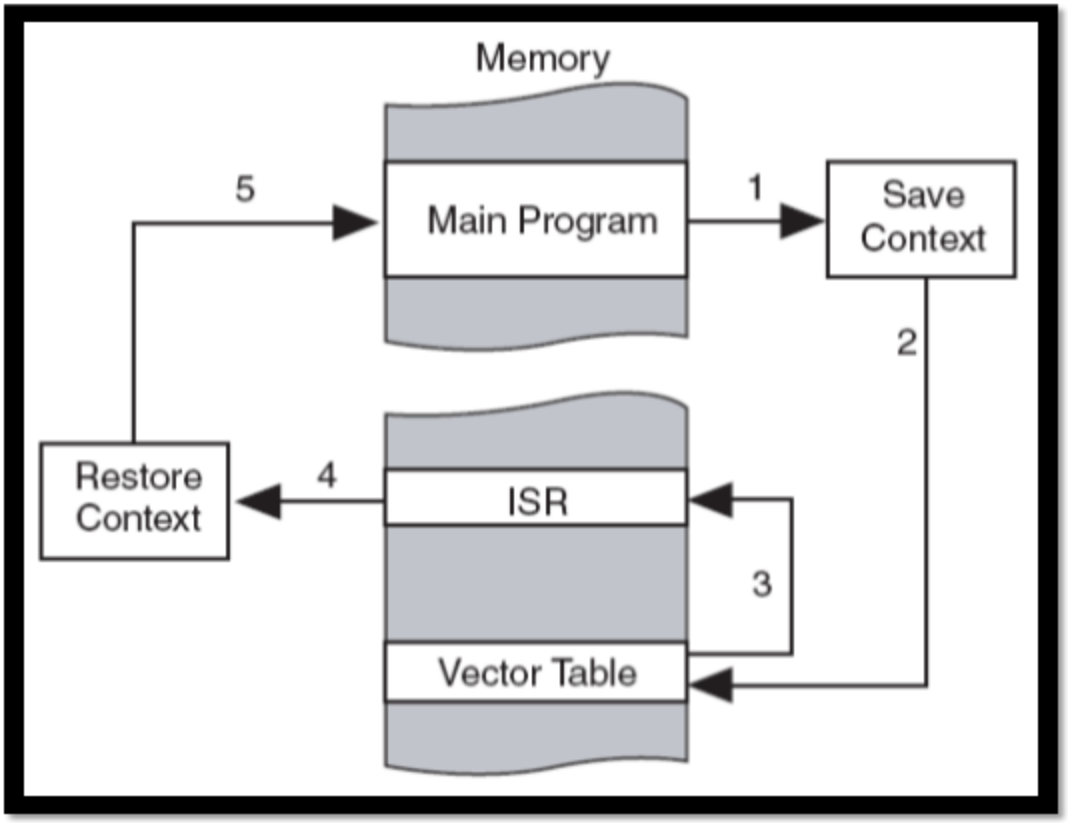
7

給兩個數 a b，判斷兩數是否互值

```
#include <stdio.h>

int gcd(int a, int b){
    return a%b==0 ? b : gcd(b, a%b);
}

int main(){
    int a,b;
    while(scanf("%d%d", &a, &b)){
        if(gcd(a,b) == 1)
            printf("互質\n");
        else
            printf("不互質\n");
    }
    return 0;
}
```

8

CPU怎麼處理interrupt ? 處理的時候會做什麼?



- Interrupt 的種類
  I. External Interrupt (外部中斷) : CPU 外的週邊元件所引起的。
    (I/O Complete Interrupt, I/O Device error)
  II. Internal Interrupt (內部中斷) :不合法的用法所引起的。
    (Debug、Divide-by-zero、overflow)
  III. Software Interrupt (軟體中斷) ：使用者程式在執行時，若需要OS 提供服務時，會藉由System Call 來呼叫OS 執行對應的service routine，完成服務請求後，再將結果傳回給使用者程式。
- Interrupt 的處理流程

Setps
1. 暫停目前process 之執行。
2. 保存此process 當時執行狀況。
3. OS 會根據Interrupt ID 查尋Interrupt vector。
4. 取得ISR (Interrupt Service Routine) 的起始位址。
5. ISR 執行。
6. ISR 執行完成，回到原先中斷前的執行。

Interrupt I/O (中斷式I/O)
其運作處理方式如下

Steps
1. 發出I/O 要求給CPU (OS) 。
2. CPU 設定I/O commands 給I/O Device controller。
3. I/O Device 運作執行。
4. PA 等待 I/O 完成。
5. PB 取得CPU 執行。
6. 當I/O 運作完成，則I/O 會發出一個「I/O Complete Interrupt」(I/O完成中斷) 通知OS。
7. OS 暫停目前process 的執行。
8. OS根據Interrupt ID 去查詢Interrupt vector，取出對應的ISR (Interrupt Service Routine) 的起始位址。
9. CPU 執行ISR。
10. ISR 執行完畢，OS 通知PA 其I/O 要求完成，將PA 的狀態改成Ready。
11. 由CPU 排班挑選process 執行。

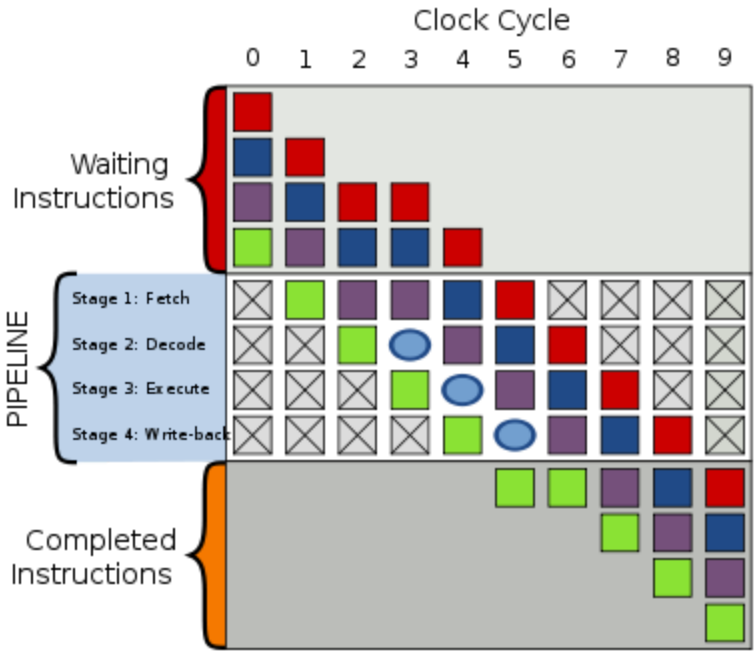
ISR:
ISR簡單來說就是中斷會跳去執行的函式,而他跟task或process不同的地方是,
做context switch的時候ISR只會PUSH部份暫存器,而task或process會push所有的暫存器

9

什麼是pipeline? 用pipeline有什麼好處/壞處?

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

IF：讀取指令，ID：指令解碼，EX：執行，MEM：記憶體存取，WB：寫回暫存器



一個氣泡(hiccup)在編號為3的時脈週期中，指令執行被延遲

是為了讓計算機和其它數位電子裝置能夠加速指令的通過速度（單位時間內被執行的指令數量）而設計的技術。
沒pipeline的架構產生的效率低，因為有些CPU的模組在其他模組執行時是閒置的。
如果一條pipeline能夠在每一個cpu clock週期接納一條新的指令，
被稱為完整管線化（fully pipelined）。
因pipeline中的指令需要延遲處理而要等待數個cpu clock，被稱為非完整管線化。
在執行相同的指令時，非pipeline的指令傳輸延遲時間（The instruction latency）比pipeline處理器明顯較
這是因為pipeline的處理器必須在資料路徑（data path）中添加額外正反器（flip-flops）

10

Deadlock

每一種資源都有一定的instances，像是可能有五個disk，不止一個的I/O devices，每一個

process 要利用資源都有以下三種階段
* 要求資源
* 使用資源
* 釋放資源

要死結必須要滿足以下四個條件
1. Mutual exclusion: 一個資源一次只能被一個process所使用
2. Hold and Wait: process取得一個資源之後等待其他的資源
3. No preemption: 資源只能由process自己釋放，不能由其他方式釋放
4. Circular wait: 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proces

針對提到的死結必須滿足的條件去做預防
Mutual exclusion: 對不可共用的資源類型而言，互斥一定成立，而可共用的資源類型，因為可以同時讀取相同檔
Hold and Wait: process必須保證一個行程在要求一項資源時，不可以佔用任何其它的資源。
No preemption: 只要某個處理元要不到所要求的資源時，便把它已經擁有的資源釋放，然後再重新要求所要資源。
Circular Wait: 確保循環式等候的條件不成立，我們對所有的資源型式強迫安排一個線性的順序。

10

atomic

atomic transaction
因為一個 transaction 由一 instruction set 所組成
如果指明了某一 transaction 是 atomic
代表這個 transaction 在執行時, 其所屬的 instruction set 不是全部執行
不然就是全部不執行, 沒有那種執行一半的

10

介紹一下Mutex、Semaphore、Spinlock

30秒：
最大的差異在於 Mutex 只能由上鎖的 thread 解鎖，
而 Semaphore 沒有這個限制，
可以由原本的 thread 或是另外一個 thread 解開。
另外，Mutex 只能讓一個 thread 進入 critical section，
Semaphore 的話則可以設定要讓幾個 thread 進入。
這讓實際上使用 Mutex 跟 Semaphore 場景有很大的差別。

_
60秒 (cont.)：
舉例而言，Mutex 的兩個特性：
一個是只能有持鎖人解鎖、一個是在釋放鎖之前不能退出的特性，
讓 Mutex 叫常使用在 critical section 只能有一個 thread 進入，
而且要避免 priority inversion 的時候；
Semaphore 也能透過 binary semaphore 做到類似的事情，
卻沒有辦法避免 priority inversion 出現。

_
120秒 (cont.)：
而 Semaphore 更常是用在同步兩個 thread 或功能上面，
因為 Semaphore 實際上使用的是 signal 的 up 與 down，
讓 Semaphore 可以變成是一種 notification 的作用，
例如 A thread 執行到某個地方時 B thread 才能繼續下去，
就可以使用 Semaphore 來達成這樣的作用。

Mutex是一把鑰匙，一個人拿了就可進入一個房間，出來的時候把鑰匙交給隊列的第一個。
一般的用法是用於串行化對critical section代碼的訪問，
保證這段代碼不會被並行的運行。
(Function前用mutex lock，執行完unction後把mutex後釋出，
sleep給其他thread使用)
(A mutex is really a semaphore with value 1.)

Semaphore是一件可以容納N人的房間，如果人不滿就可以進去，
如果人滿了，就要等待有人出來。
對於N=1的情況，稱為binary semaphore。
一般的用法是，用於限制對於某一資源的同時訪問。

semaphore, mutex 會有睡覺的副作用,
什麼是「睡覺」, 就是原本在執行的這段程式碼,
在執行了 semaphore, mutex 的 function 之後,
有可能會自己把 cpu 讓出去 (有點像是 coroutine 的 yeild), 要
等一會兒才會再繼續執行;
那 spinlock 呢? 答案很有趣, 在 kernel space 不會,
甚是在 kernel space 的 spinlock 還需要關閉中斷,
很多情境都比 user mode 複雜;
在 user space 一樣會讓出 cpu,
不過是被迫讓出去 (os 排程的管理), 而不是自己讓出去

spinlock
spinlock利用test and set這個指令看有沒有辦法取得lock
因為是指令層級的操作所以有辦法達到atomic
當lock無法取得時會用polling的方式不斷嘗試
特別的地方是當他取得lock時 process將不會進入睡眠（沒有context switch）
效能會比semaphore好 因為他不做context switch可以一直執行

mutex vs spinlock
Mutex 屬於 sleep-waiting 類型的鎖。
Spin lock 屬於 busy-waiting 類型的鎖。
只能用在 thread 程式, 如果是兩個 process,
你沒辦法用 mutex/spinlock 去保護共享資源,
共享資源並非只有共享記憶體/變數, file, 硬體資源都是。

mutex vs semaphore
mutex 除了擁有者外還有優先權的概念, 類似 thread 的優先權那樣
semaphore 不只能用在 thread 程式, 兩個不同的 process 也可以用 semaphore 共享資源。

不過 3 者的共同點都需要 atomic 的操作。

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

pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* trythis(void *arg)
{
    pthread_mutex_lock(&lock);

    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for(i=0; i<(0xFFFFFFFF);i++);

    printf("\n Job %d has finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
}

int main(void)
{
    int i = 0;
    int error;

    if (pthread_mutex_init(&lock, NULL) != 0)
    {
        printf("\n mutex init has failed\n");
        return 1;
    }

    while(i < 2)
    {
        err = pthread_create(&(tid[i]), NULL, &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created :[%s]", strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

```
/**
Job 1 started
Job 1 finished
Job 2 started
Job 2 finished
**/
```

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>

sem_t semaphore; // 旗標
int counter = 0;

// 子執行緒函數
void* child() {
  for(int i = 0;i < 5;++i) {
    sem_wait(&semaphore); // 等待工作
    printf("Counter = %d\n", ++counter);
    sleep(1);
  }
  pthread_exit(NULL);
}

// 主程式
int main(void) {

  // 初始化旗標，僅用於本行程，初始值為 0
  sem_init(&semaphore, 0, 0);

  pthread_t t;
  pthread_create(&t, NULL, child, NULL);

  // 送出兩個工作
  printf("Post 2 jobs.\n");
  sem_post(&semaphore);
  sem_post(&semaphore);
  sleep(4);

  // 送出三個工作
  printf("Post 3 jobs.\n");
  sem_post(&semaphore);
  sem_post(&semaphore);
  sem_post(&semaphore);

  pthread_join(t, NULL);

  return 0;
}
/**
Post 2 jobs.
Counter = 1
Counter = 2
Post 3 jobs.
Counter = 3
Counter = 4
```

```
Counter = 5
**/
```

Pi	Pi	Pi

repeat
A;
signal(S1);
wait(S3);
unitl False
	repeat
	wait(S1);
	B
	signal(S2);
	unitl False
		repeat
		wait(S1);
		C
		signal(S3);
		unitl False

11

System Call

System call 是 process 與作業系統之間的介面。
System call 是由 Linux kernel 所實作並提供給使用者，
user-space program 可透過 system call 與Linux kernel 溝通
作為user program執行時與OS之間的溝通介面，當user program需要OS提供服務時，
藉由呼叫system call(伴隨trap to monitor mode)通知OS，
OS可依據system call ID查表，啟動service routine執行，
得到結果，再傳回給user program，完成服務請求

System Call種類：(P3-8~3-9)
Process Control
File Management
Device Management
Information Maintenance
Communication

行程控制(Process Control)系統呼叫
如前述，CPU 抓取主記憶體(Main Memory)的程式(Program)並執行其工作(Job)如此是謂“行程(Process)”。
執行過程中，凡遇行程衝撞系統的動作，均須作系統呼叫，
由具有安全性的即定程序引導執行。行程作系統呼叫的項目有：
1. 行程的起動與終止：
(a)呼叫作業系統執行載入(Load)，將程式指令從主記憶體抓取至 CPU 執行(Execute)之；
(b)呼叫作業系統執行建立行程(Create Process)，執行完畢後，呼叫作業系統執行終止行程(Terminate Proces
(c)在正常情況下，呼叫作業系統執行終止程式(End Program)，在有錯誤的情況下，摒棄程式(Abort Program)，

2. 記憶體分配 (Allocate and Free Memory)：
當執行行程時，須配合主記憶體的空間、或 CPU 內的暫存器作資料(Data)儲存，
這些動作均須呼叫作業系統執行記憶體分配(Allocate)、或釋出記憶體(Free)。

3. 行程屬性 (Process Attributes)：
在執行過程中，為了配合其他行程的需要，往往要了解其他行程的屬性，
此時呼叫作業系統讀取其他行程的屬性(Get Process Attributes)；
亦或呼叫系統設定本身行程的屬性(Set Process Attributes)，以供其他行程觀察使用。

4. 行程等待 (Wait)：
在執行過程中，有許多情況需要行程(Process)作等待，
等待進入 CPU、等待滑鼠事件、等待鍵盤事件等等，
為了執行過程井然有序，呼叫作業系統執行行程等待。

檔案管理(File Management)系統呼叫
一組資料(Data)或一個程式(Program)儲存於連續的記憶體內，
以一個名稱代表之，是謂“檔案(File)”。
在記憶體建立檔案、存取檔案等等作為，均須作系統呼叫由作業系統導引執行，
其中項目有：
1. 檔案之建立與刪除(Create / Delete File)：建立或刪除一個檔案，均會改變記憶體的使用情況，必須呼叫作
2. 檔案資料存取 (Read / Write)：存取檔案資料必經之過程為：
(a)開啟檔案(Open File)、(b)存取檔案(Read / Write)、(c)關閉檔案(Close File)。
為了安全，均須呼叫作業系統導引執行。
3. 檔案屬性 (File Attributes)：為了配合存取檔案的需要，往往需要了解其他檔案的屬性，此時呼叫作業系統

裝置管理(Devices Management)系統呼叫
電腦系統之各項硬體資源與週邊設備均歸類為裝置(Device)，
當行程執行時將會要求分配資源、及存取資料，這些都是牽動系統的行為，須作系統呼叫執行：
1. 裝置之需求與釋放 (Request / Release Device)：在電腦系統中，裝置(Device)即是一種資源，當行程(Pr
2. 裝置資料存取 (Read / Write)：存取裝置資料是 I/O 行為，為了安全，須呼叫作業系統導引執行。
3. 裝置屬性 (Device Attributes)：為了配合裝置執行 I/O 的需要，往往需要了解裝置的屬性，此時呼叫作業系
4. 邏輯連接 (Logically Attach / Detach)：於程式(Program)內設定一個名稱(Name)，比擬為某境外裝置，

資料維護(Information Maintenance)系統呼叫
隨著時間的改變，將資料更新是謂“資料維護(Information Maintenance)”，如果是系統資料的更新，為了安全，
1. 設定時間或日期(Set Time or Date)：電腦內部有計時裝置，配合計時器的運轉，作業系統對映顯示時間及日期
2. 存取系統資料 (Get / Set System Data)：作業系統的資料牽涉電腦的整體運作，故其任何更新之改變需呼叫
3. 存取行程、檔案、或裝置之屬性 (Get / Set Process、File、or Device Attributes)：如前述，行程、檔

連線通訊(Communication)系統呼叫
分散式系統是將散置各處的電腦以連線連通，執行訊息傳遞等 I/O 存取行為，故須作系統呼叫執行：
1. 建立或中斷連通連線 (Create / Delete Communication Connection)：實體連線的連接屬於網路實體層，如
2. 輸入輸出網路資料 (Send / Receive Messages)：網路資料的輸入輸出牽涉甚多，除了 I/O 機制外，還有網
3. 狀態資訊之轉換 (Transfer Status Information)：為了配合不同的環境條件，系統須對某些區塊設定狀態旗
4. 使用遠端裝置 (Attach / Detach Remote Devices)：當使用網路遠端其他電腦或裝置時，系統應有相對之使

12

OSI模型

第7層應用層（Application Layer）
　　提供為應用軟體而設的介面，以設定與另一應用軟體之間的通訊。例如：HTTP，HTTPS，FTP，TELNET，SS
第6層表達層（Presentation Layer）
　　把資料轉換為能與接收者的系統格式相容並適合傳輸的格式。
第5層會議層（Session Layer）
　　負責在資料傳輸中設定和維護電腦網路中兩台電腦之間的通訊連接。
第4層傳輸層（Transport Layer）
　　把傳輸表頭（TH）加至資料以形成資料包。傳輸表頭包含了所使用的協定等傳送資訊。例如：傳輸控制協定
第3層網路層（Network Layer）
　　決定資料的路徑選擇和轉寄，將網路表頭（NH）加至資料包，以形成封包。網路表頭包含了網路資料。例如
第2層資料鏈結層（Data Link Layer）
　　負責網路尋址、錯誤偵測和改錯。當表頭和表尾被加至資料包時，會形成了影格。資料鏈表頭（DLH）是包含
　　分為兩種子層：logic link control sublayer & media access control sublayer
第1層實體層（Physical Layer）
　　在局部區域網路上傳送影格，它負責管理電腦通訊裝置和網路媒體之間的互通。包括了針腳、電壓、線纜規

13

Real-time operating system, RTOS

實時作業系統與一般的作業系統相比，最大的特色就是其「實時性」，
也就是說，如果有一個任務需要執行，
實時作業系統會馬上（在較短時間內）執行該任務，
不會有較長的延時。這種特性保證了各個任務的及時執行
設計實時作業系統的首要目標不是高的吞吐量，而是保證任務在特定時間內完成

14

3 / 4 way handshake

[SYN] 我: 哈嚕, 有沒有聽到聲音?

[SYN, ACK] A: 有聽到，那我呢?有沒有聽到聲音

[ACK] 我: 有滴~

[FIN, ACK]
[ACK]
是四向交握協定Four-way Handshake，用來關閉連線的

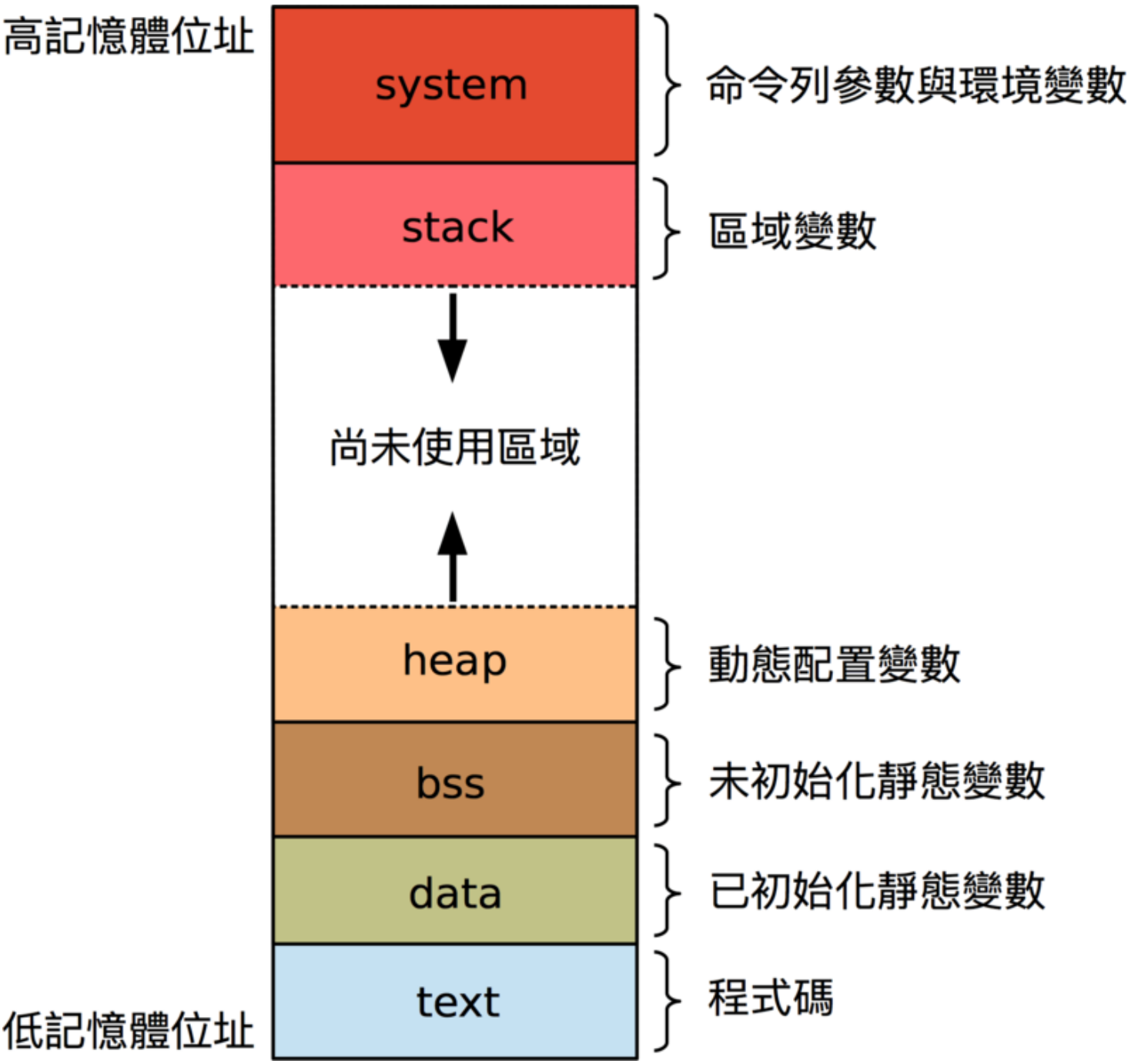
1. (B) --> ACK/FIN --> (A)

2. (B) <-- ACK <-- (A)

3. (B) <-- ACK/FIN <-- (A)

4. (B) --> ACK --> (A)

stack/heap overflow



stack：區域變數
堆疊區段（stack segment）用於儲存函數的區域變數，以及各種函數呼叫時需要儲存的資訊（例如函數返回的記憶
區域變數(local variable)、函式參數(function/method parameter)、函數的返回位址(function/method r

```
int foo()
{
    return foo();  //這裡出現自我呼叫
}
```

heap：動態配置變數
heap 區段的記憶體空間用於儲存動態配置的變數，例如 C 語言的 malloc 以及 C++ 的 new 所建立的變數都是儲
堆疊區段一般的狀況會從低記憶體位址往高記憶體位址成長，而 heap 剛好從對面以相反的方向成長。

Heap中的資料如果沒有正常的回收，將會逐步成長到將記憶體消耗殆盡

18

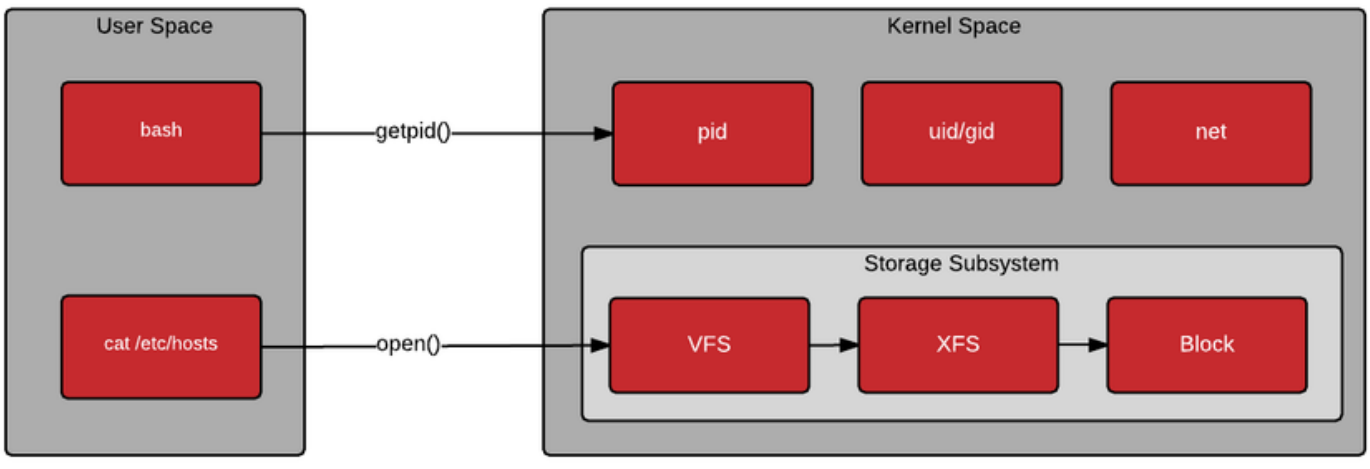
Memory Hierarchy

Register – Cache – Main Memory – Disk –Tape
愈快 ← 速度 → 愈慢
愈昂貴 ← 價格 → 愈便宜
愈小 ← 容量 → 愈大

SRAM (Static RAM)：用Flip/Flop儲存，速度快，密度低(元件大)，成本高，作Cache等快速記憶體，不須Refre
DRAM (Dynamic RAM)：用電容器製作，速度慢，密度高(元件小)，成本低，為Main Memory的主體，須Refresh
儲存的電荷會隨著時間漸漸消失，因此需要有個再充電（Refresh）的動作保持電容儲存的資料

17

User space/ Kernel space



簡單說，Kernel space 是 Linux 內核的運行空間，User space 是用戶程序的運行空間。
為了安全，它們是隔離的，即使User的程序崩潰了，內核也不受影響。
Kernel space 可以執行任意命令，調用系統的一切資源；
User space 只能執行簡單的運算，不能直接調用系統資源，必須通過系統接口（又稱 system call），才能向內核

```
str = "my string" // 用戶空間
x = x + 2
file.write(str) // 切換到內核空間

y = x + 4 // 切換回用戶空間
```

18

```
prefix / postfix / infix
```

PreFix（前序式）：* + 1 2 + 3 4

InFix（中序式）： (1+2)*(3+4)

PostFix（後序式）：1 2 + 3 4 + *

PostFix:將中序式轉換為後序式 的好處是，不用處理運算子先後順序問題，只要依序由運算式由前往後讀取即可。
用手算的方式來計算後序式相當的簡單，將運算子兩旁的運算元依先後順序全括號起來，然後將所有 的右括號取代為

```
a+b*d+c/d   =>    ((a+(b*d))+(c/d)) -> abd*+cd/+
```

C code: 一個一個讀入stack，讀到右括號，把stack pop

```
#include <stdio.h>
#include <vector>
#include <iostream>
#include <string.h>

using namespace std;
vector<char> mystack;

char input[100];

int main(){
    while(scanf("%s", input)){
        mystack.clear();
        for(int i=0; i<strlen(input); i++){
            if(input[i]>= '0' && input[i] <= '9')
                printf("%c", input[i]);
            else if(input[i] == ')'){
                printf("%c", mystack.back());
                mystack.pop_back();
            }
            else if(input[i] != '(')
                mystack.push_back(input[i]);
        }
        while(mystack.empty()!=true){
            printf("%c", mystack.back());
            mystack.pop_back();
        }
        printf("\n");
    }
    return 0;
}
```

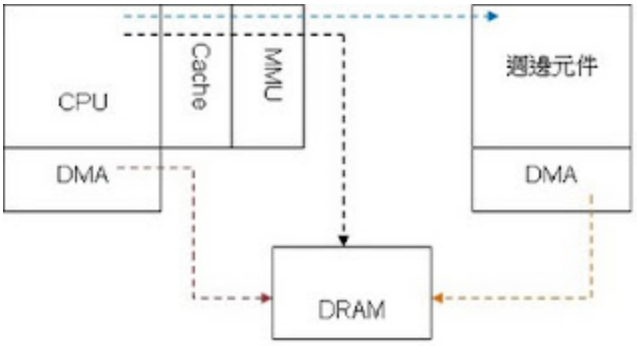
18

當場寫出一個反轉字串的程式，輸入是一個char*

```
#include <stdio.h>
#include <vector>
#include <iostream>

using namespace std;

vector<char> mystack;
char input[100];

void str_reverse(char* input){
    mystack.clear();
    while(*input != '\0'){
        mystack.push_back(*input);
        input++;
    }
    while(mystack.empty() != true){
        printf("%c", mystack.back());
        mystack.pop_back();
    }
    printf("\n");
}

int main(){
    while(scanf("%s", input)){
       str_reverse(input);
    }
    return 0;
}
```

19

DMA是什麼，好處是？請簡單在白板上畫出他的架構圖。



直接記憶體存取（Direct Memory Access，DMA）
是電腦科學中的一種記憶體存取技術。
它允許某些電腦內部的硬體子系統（電腦外設），
可以獨立地直接讀寫系統記憶體，而不需中央處理器（CPU）介入處理 。
在同等程度的處理器負擔下，DMA是一種快速的資料傳送方式。
很多硬體的系統會使用DMA，包含硬碟控制器、繪圖顯示卡、網路卡和音效卡。

DMA是所有現代電腦的重要特色，它允許不同速度的硬體裝置來溝通，
而不需要依於中央處理器的大量中斷負載。
否則，中央處理器需要從來源把每一片段的資料複製到暫存器，
然後把它們再次寫回到新的地方。
在這個時間中，中央處理器對於其他的工作來說就無法使用。

20

封裝、繼承、多型

封裝、繼承、多型為物件導向三大基礎 。
此三者具有次序性 ， 沒有封裝就不可能有繼承 、沒有繼承就不可能有多型。

封裝 (Encapsulation) 的目的
是將 Class 裡的屬性用 private 隱藏，
只能透過public的方法存取資料。
(隱藏程式細節，避免直接處理造成的困擾。使開發與維護更容易)
拿現實世界來舉例的話
自動販賣機來說好了，有投錢的裝置，選擇商品的按鈕，拿取商品的箱子
買可樂的步驟投錢，選可樂，拿可樂
你不用知道為何投錢後按鈕燈會亮，按按鈕後可樂如何掉下來
這些知識都被封裝起來了

繼承 (Inheritance) 的目的，是
要達到「程式碼再用」(Code Reuse) 或「介面再用」。
透過繼承，可以適當的切割類別，
並在衍生類別中重複使用、擴充和修改基底類別中定義的行為，
又不破壞原先基底類別設計。

多型 (Polymorphism) 指的是不同型態的物件，
定義相同的操作介面，由於被呼叫者 (Caller)有著相同的介面，
呼叫者並不用指定特別型別，只需針對介面進行操作，
實際執行的物件則在runtime決定，藉此增加程式碼的彈性。

```
class Drink{}
class Cola entends Drink{}
class Juice entends Drinks{}
Drink mydrinks[] = [ new Cola(), new Juice() ]
```

21

以一個單字為單位 反轉一個輸入字串

```
#include <stdio.h>
#include <vector>
#include <iostream>
#include <string.h>

using namespace std;

vector<char*> mystack;
char input[100];

int main(){
    while(gets(input) != NULL){
       char* str_ptr = strtok(input," ");
       while(str_ptr != NULL){
            mystack.push_back(str_ptr);
            str_ptr = strtok(NULL, " ");
       }

       while(mystack.empty() != true){
            printf("%s ", mystack.back());
            mystack.pop_back();
       }
       printf("\n");

    }
    return 0;
}
```

22

求兩個NxN矩陣的乘積

```
#include <stdio.h>
#include <iostream>

using namespace std;

int m1[3][3] = {1,2,3,
                4,5,6,
                7,8,9};
int m2[3][3] = {2,2,2,
                3,3,3,
                4,4,4};

int main(){
    int ans[3][3];
    for(int i =0; i<3; i++){
        for(int j=0; j<3; j++){
            int temp = 0;
            for(int k=0; k<3; k++){
                temp += m1[i][k] * m2[k][j];
            }
            printf("%d ", temp);
        }
        printf("\n");
    }
    return 0;
}
```

23

質數表

```
#include <stdio.h>
#include <iostream>
#include <math.h>
#include <vector>

using namespace std;
vector<int> myprime;

int prime[1001];
int main(){
    for(int i=2; i<=sqrt(1000); i++){
        for(int j=i*i; j<=1000; j=j+i){
            prime[j] = 1;
        }
    }
    for(int i=2; i<=1000; i++){
        if(prime[i]==0){
            myprime.push_back(i);
            printf("%d ", i);
        }
    }
    return 0;
}
```

24

10進位轉3進位

```
#include <stdio.h>
#include <iostream>
#include <math.h>
#include <vector>

using namespace std;
vector<int> ans;
int main(){
    int input;
    while(scanf("%d", &input)){
        while(input != 0){
            ans.push_back(input%3);
            input = input/3;
        }
        while(ans.empty()!=true){
            printf("%d", ans.back());
            ans.pop_back();
        }
    }

    return 0;
}
```

25

C Pointer

```
int a[] = {1, 2, 3, 4, 5, 6};
    int *p = a;
    *(p++) += 100;
    *(++p) += 100;

    for(int i=0; i<6; i++){
        printf("%d ", a[i]);
    }

// 101 2 103 4 5 6
```

++ 在後 優先權最低　先取值 +=100 再 指標後移。
++ 在前 優先權最高　先指標後移 再取值 +=100。

```
char s[] = "0113256";
char *p = s;

printf("%c",*p++);
printf("%c",*(p++));
printf("%c",(*p)++);
printf("%c",*++p);
printf("%c",*(++p));
printf("%c",++*p);
printf("%c",++(*p));

printf("\n");


printf(s);
```

第一行字串為 : 0113234
第二行字串為 : 0123456

*p++ = *(p++) 先取值 後指標下移
*++p = *(++p) 先指標下移 後取值
++*p = ++(*p) 先值+1 後取值
(*p)++ 先取值 後值+1

*p = 取值
p = 取指標位址
++在前 = 先加1再取值
++在後 = 先取值後加1

& :取變數在記憶體裡面的位置
* :取變數在記憶體裡面的值

ask: the value of *(a+1), (*p-1)?

```
int a[5] ={1,2,3,4,5};
int *p = (int *)(&a+1);

// 2  6946571
```

26

3的倍數check

1. 使用加減乘除

```
(n / 3 * 3 == n) ? yes : no
```

2. 不能用乘除

```
Explain:
4 = 3 + 1
4^2 = (3+1)^2 = 3^2 + 2*3 + 1 = 3*n + 1
4^3 = 3*m + 1
```

```
int input = 12;
int remainder = 0;
while(input != 0){
  remainder += input & 0x3;
  input = input>>2;
  if(input == 0 && remainder >= 3){
    input = remainder-3;
    remainder = 0;
  }
}
```

27

不使用暫存變數交換兩個變數 (Swap two variables without using a temporary variable)

```
a = a ^ b
b = a ^ b
a = a ^ b
```

28

quick sort

```
void quickSort( vector< int > &num, int L, int R ) {
        // if seqence is empty, return
        if ( L == R )
                return;
        // put all elements less than pivot to left, and others to right
        int pivot = num[ R - 1 ], pos = L;
        for ( int i = L; i < R - 1; ++i )
                if ( num[ i ] < pivot )
                        swap( num[ i ], num[ pos++ ] );
        // put pivot between left part and right part
        swap( num[ R - 1 ], num[ pos ] );
        // sort sub-seqence
        quickSort( num, L, pos );
        quickSort( num, pos + 1, R );
}

int main() {
        int n;
        while ( ~scanf( "%d", &n ) ) {
                int x;
                vector< int > num;
                for ( int i = 0; i < n; ++i ) {
                        scanf( "%d", &x );
                        num.push_back( x );
                }

                quickSort( num, 0, num.size() );
                for ( int i = 0; i < num.size(); ++i )
                        printf( "%d ", num[ i ] );
                puts( "" );
        }
        return 0;
}
```

28

從兩個數字中找出最大的一個而不使用判斷描述

```
int max(int a, int b)
{
    int diff = a - b;
    int sign_bit= unsigned(diff) >> (sizeof(int) * 8 - 1);
    int array[] = {a, b};
    return array[sign_bit];
}
```

或

```
int max(int a, int b)
{
    return ((a + b) + abs((a - b))) / 2;
}
```

29

merge sort

```
void merge(int left, int right){
    int i, j, k;
    int mid = (left + right) / 2;

    for (k = 0, i = left, j = mid+1 ; i <= mid || j <= right ; k++){
        if (i > mid)
            tmp[k] = num[j++];
        else if (j > right)
            tmp[k] = num[i++];
        else if (num[i] < num[j])
            tmp[k] = num[i++];
        else
            tmp[k] = num[j++];
    }
    for (i = left, k = 0; i <= right; i++, k++)
        num[i] = tmp[k];
}

void mergeSort(int left, int right){
    int mid = (left + right) / 2;

    if (left < right){
        mergeSort(left, mid);
        mergeSort(mid+1, right);
        merge(left, right);
    }
}
int main(void){
    mergeSort(0, 9);
    return 0;
}
```

30

pointer and &

```
int a = 3;
int &c= a;
printf("%d\n", c);
c *= a;
printf("%d\n", a);
printf("%d\n", c);
// 3 9 9
// c就是指到a的位置 簡單來說c完全等於a 不是只有得到a的值
```

31

Function Pointer

例如三個副函式：

```
(1) int add(int);
(2) float add2(float);
(3) int add3(int,int);
```

並且宣告一個函式指標：

```
int (*pf)(int);
```

則：

```
(a.) pf = add;     //正確
(b.) pf = add2;   //錯誤，參數資料型態不匹配
(c.) pf = add3;   //錯誤，引入參數個數不匹配
```

```
int do_math(float arg1, int arg2) {
    return arg2;
}

int call_a_func(int (*call_this)(float, int)) {
    int output = call_this(5.5, 7);
    return output;
}

int final_result = call_a_func(do_math);
```

```
typedef int (*MathFunc)(float, int);

int do_math(float arg1, int arg2) {
    return arg2;
}

int call_a_func(MathFunc call_this) {
    int output = call_this(5.5, 7);
    return output;
}

int final_result = call_a_func(do_math);
```

32

ifndef /define/ endif 作用和用法

ifndef/define/endif」主要目的是防止頭文件的重複包含和編譯

用法：

.h文件，如下：

```
#ifndef XX_H
#define XX_H

...

#endif
```

33

Explain "struct" "union" "enum"

```
struct Employee{
        char name[30]; // 名字
        int age; //年齡
        char gender; // 性別，'M' or 'F'
        double salary; // 薪水
};
struct Employee employee; // 宣告變數employee，記得前面要加struct
```

```
union Var{
        char ch;
        int num1;
        double num2;
};
int main(void) {
        union Var var = {'x'}; // 初始化只能指定第一個成員
        // union Var var = {123}; 這句是不行的

        printf("var.ch = %c\n",var.ch);
        printf("var.num1 = %d\n",var.num1); // 內容是無效的
        printf("var.num2 = %.3f\n\n",var.num2); // 內容是無效的

        var.num1 = 123;

        printf("var.ch = %c\n",var.ch); // 內容是無效的
        printf("var.num1 = %d\n",var.num1);
        printf("var.num2 = %.3f\n\n",var.num2); // 內容是無效的

        var.num2 = 456.789;

        printf("var.ch = %c\n",var.ch); // 內容是無效的
        printf("var.num1 = %d\n",var.num1); // 內容是無效的
        printf("var.num2 = %.3f\n\n",var.num2);

        return 0;
}
```

是一組由識別字所代表的整數常數

除非特別指定，不然都是由0開始，接下來遞增1，例如以下語法：

```
enum week{Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday};
```

以上從Sunday開始，各個識別字被依序設定為0到6，你也可以指定數值

```
enum week{Monday=1,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday};

int main(void)
{
        enum week w;
        const char *day_name[] = {
                "","Monday","Tuesday","Wednesday",
                "Thursday","Friday","Saturday","Sunday"
        };

        for(w=Monday; w <= Sunday; w++)
                printf("%s\n",day_name[w]);

    return 0;
}
```

34

volatile

```
 i = *pPort;
 j = *pPort;
 k = *pPort;
```

以上的i, j, k很有可能被compiler最佳化而導致產生
`i = j = k = *pPort;`
的code, 也就是說只從pPort讀取一次, 而產生 `i = j = k` 的結果, 但是原本的程式的目
的是要從同一個I/O port讀取3次的值給不同的變數, i, j, k的值很可能不同(例如從此
I/O port 讀取溫度), 因此`i = j = k`的結果不是我們所要的

一個參數可以同時是const也是volatile嗎? 解釋為什麼。
.是的。舉的例子是"只讀的狀態暫存器"。
它是volatile因為它可能被意想不到地改變。它是const因為程式不應該試圖去修改它。

35

#error

#error : 在編譯時，輸出錯誤訊息，警告使用者某些錯誤，並且不會真的進行編譯，在巨集處理階段就會停止。

```
#if !defined( HELLO_MESSAGE )
   # error "You have forgotten to define the header file name."
#endif
```

36

lvalue and rvalue

Lvalue: 就是一個運算式後還保留其狀態的一個物件 就是Lvalue; 也
就是說 所有的變數(variables)包含nonmodifiable, const 的變數都是Lvalue.
這邊一個重點是 const的變數也是Lvalue

Rvalue: 就是一個運算式過後其狀態就不會被保留了,
也就是一個暫存的數值

另一種說法(非完全正確, 但是可以依此來稍做判斷)
能出現在assignment 運算子(=)的左邊的稱為Lvalue, 而只能出現在右邊的稱為Rvalue

這邊只有說出現在左邊的是Lvalue, 但沒說Lvalue不能出現在右邊, 因此Lvalue在=運算子的左右兩邊都是被允許的
而Rvalue是不能出現在左邊的; 這邊有沒有注意到, Lvalue是可以被放到右邊的, 也就是說Lvalue也可以被當作R

37

Synchronous call 跟 Asynchronous call

Async是接收到需求，不用一直等到需求完成再執行其他需求(0)
Async與Sync的差別在於：發送需求的人是否需要等到需求完成才可以執行其他事情。

38

pointer-to-pointer

```
//改變外來的pointer的值(非pointer所指向的變數)
int gint = 0;
void changePtr (int *pInt)
{
    pInt = &gint;
}
void main ()
{
    int local_int = 1;
    int *localPtr = &local_int;
    changePtr (localPtr);
    printf ("%d\n", *localPtr);
}
```

上述例子, 印出來的數值仍為1,
因為changePtr的pInt是localPtr的複本, 對pInt做變更, 其實並不會影響到localPtr本身

使用call by pointer (or address)來傳遞參數,
被呼叫的函式只是複製pointer的值過去罷了!
所以當我們想在函式內改變外來的pointer的值(非pointer所指向的變數),
且函式外部能使用其改變的pointer的值,
此刻就是call by pointer to pointer登場的最佳機會!

39

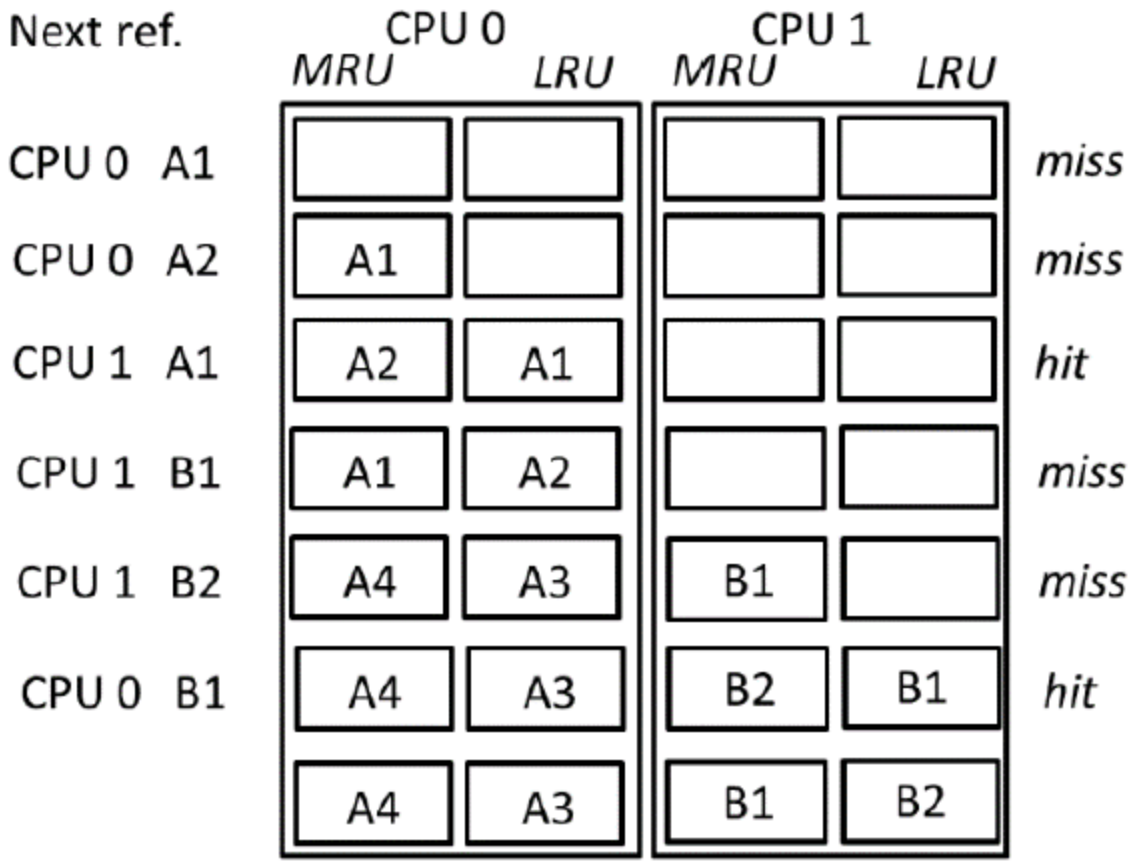
Virtual Function

通常你再寫base class時，
你的member function可能在其他class 繼承 base class時，
該函式的會被重新實作，這時候，你的base class的那個會被子class重新實作的成員
函式就要宣告成virtual。

```
// 假設這三個類別都繼承 GradeList:
GradeSheet *pSheet = new GradeSheet();
OtherSheet *pOther = new OtherSheet();
SpecialSheet *pSpecial = new SpecialSheet();
...其他衍生類別的 instance...

// 用父類別指標陣列來裝子類別instances 的指標:
GradeList * pList[10];
pList[0] = pSheet;
pList[1] = pOther;
pList[2] = pSpecial;
pList[3] = ...;

// 假設校務系統的設計是呼叫所有班級的分數調整函式:
for (int i=0; i<10; ++i)
{
    pList[i]->adjust_grade_list();
}
```

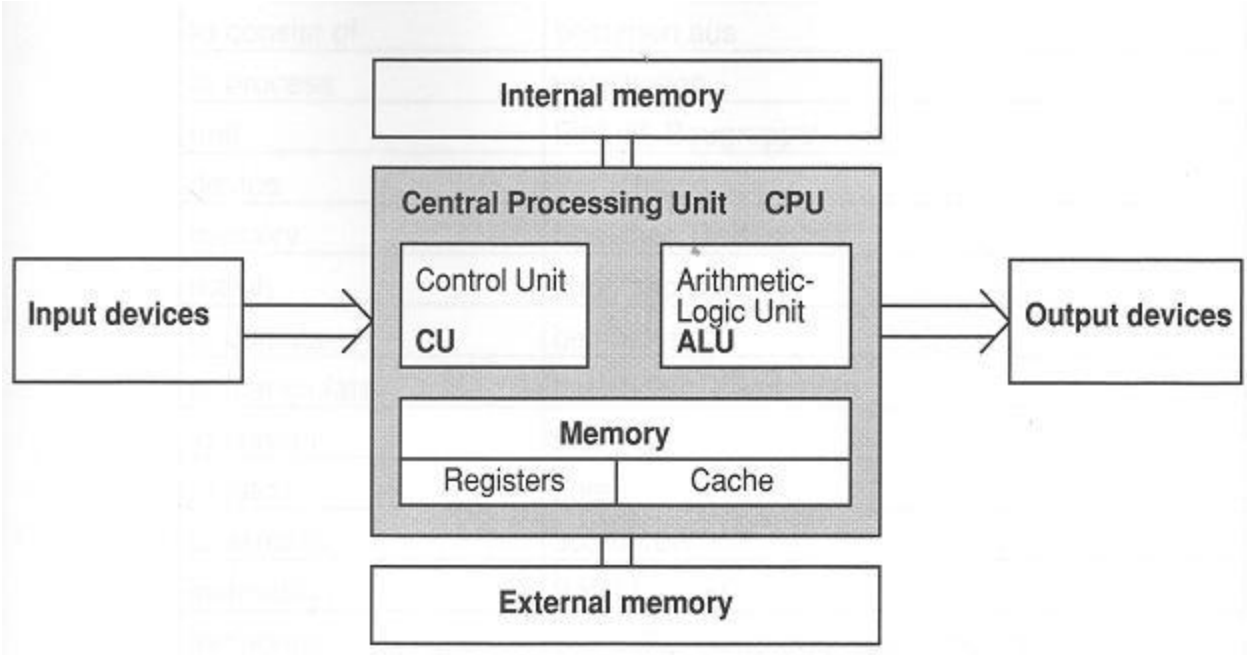
40

cache algorithm



41

CPU架構



中央處理單元(CPU)主要由運算器、控制器、寄存器三部分組成，
從字面意思看運算器就是起著運算的作用，
控制器就是負責發出CPU每條指令所需要的信息，
寄存器就是保存運算或者指令的一些臨時文件，這樣可以保證更高的速度。
Control Unit
CPU 可以說是電腦的大腦，而 Control Unit 則可以說是CPU的艦長，負責發號司令負責叫ALU運算跟控制暫存器
Arithmetic Logic Unit(ALU)
負責實際運算的部份，Input 由 Control Unit 控制，運算完的結果透過 Output 或是 Flags 來讓 Control U
Register
Register(暫存器)在 CPU 裡面扮演重要的腳色，它們讓 ALU 計算出來的數值可以有地方暫時存放，以供之後的運
其它的Register
Instruction register: 把指令的種類輸出到 Control Unit
Instruction address register: 下一個指令所在的 address 輸出到 memory address register
Memory address register: 把 address 輸出到 Memory

因為多個 Register 彼此共用一個 bus，為了要讓 ALU 可以讀到兩個Input，會在其中一個Input 加入一個 Tem

Intel和ARM處理器的第一個區別是，前者使用複雜指令集(CISC)，而後者使用精簡指令集(RISC)

開始的處理器都是CISC架構，為了減少程式設計師的設計時間，逐漸開發出單一指令，複雜操作的程式碼，設計師只

RISC的優點列舉如下：
指令長度固定，方便CPU解碼，簡化解碼器設計。
盡量在CPU的暫存器（最快的記憶體元件）裡操作，避免額外的讀取與載入時間。
由於指令長度固定，更能受益於執行線路管線化（pipeline）後所帶來的效能提升。
處理器簡化，電晶體數量少，易於提升運作時脈。比起同時脈的CISC處理器，耗電量較低。

RISC的缺點列舉如下：
複雜指令需要由許多的小指令去完成，程式變得比較大，記憶體也占用比較多，這在硬碟昂貴，常常使用磁帶儲存的
程式變長，代表著讀取工作變得繁重，需要更多的時間將指令從記憶體載入至處理器內。

42

Bitwise

判斷一個整數是不是 2 的次方

```
bool ispow2(int n)
{
    return (n & -n) == n;
}
```

整數加一與減一

```
// 注意：比直接加一和減一還要慢。
void add_one(int& x)
{
    return -~x; // ++x
}

void sub_one(int& x)
{
    return ~-x; // --x
}
```

整數變號

```
void negative(int& x)
{
    return ~x + 1;          // -x;
}
void negative(int& x)
{
    return (x ^ -1) + 1;    // -x;
}
```

Reverse

```
0xaaaaaaaa = 10101010101010101010101010101010
0x55555555 = 01010101010101010101010101010101
0xcccccccc = 11001100110011001100110011001100
0x33333333 = 00110011001100110011001100110011
0xf0f0f0f0 = 11110000111100001111000011110000
0x0f0f0f0f = 00001111000011110000111100001111
0xff00ff00 = 11111111000000001111111100000000
0x00ff00ff = 00000000111111110000000011111111

x = (((x & 0xaaaaaaaa)>> 1) | ((x & 0x55555555) <<1));
x = (((x & 0xcccccccc)>> 2) | ((x & 0x33333333) <<2));
x = (((x & 0xf0f0f0f0)>> 4) | ((x & 0x0f0f0f0f) <<4));
x = (((x & 0xff00ff00)>> 8) | ((x & 0x00ff00ff) <<8));
return((x>> 16) | (x <<16));
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
(2 1) (4 3) (6 5) (8 7) (10 9) (12 11) (14 13) (16 15)
(4 3 2 1) (8 7 6 5) (12 11 10 9) (16 15 14 13)
(8 7 6 5 4 3 2 1) (16 15 14 13 12 11 10 9)
(16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1)
```

幾個1(每次減少一個1)

```
int bitcount(unsigned int n)
{
   int count = 0 ;
   while (n)
   {
      count++ ;
      n &= (n - 1) ; //關鍵演算之處
   }
   return count ;
}
```

43

通訊概論

```
FDMA (Frequency Division Multiple Access)
TDMA (Time Divison Multiple Access)
CDMA (Code Division Multiple Access )
CDMA又稱碼分多址，是在無線通訊上使用的技術，CDMA允許所有的使用者同時使用全部頻帶(1.2288Mhz)，並且把其
CDMA，就是利用展頻的通訊技術，因而可以減少手機之間的干擾，並且可以增加用戶的容量，而且手機的功率還可以
```

傅立葉轉換

傅立葉變換將函數的時域（紅色）與頻域（藍色）相關聯。頻譜中的不同成分頻率在頻域中以峰值形式表示

傅里葉級數(Fourier Series)的頻譜

傅里葉級數的本質是將一個週期的信號分解成無限多分開的（離散的）正弦波
在這幾幅圖中，最前面黑色的線就是所有正弦波疊加而成的總和，也就是越來越接近矩形波的那個圖形。

44

write through / write back 的差別，哪個快？

write through: 是直接把資料寫回記憶體
write back: 是先將資料按一定數量接受下來，然後將相同位址的資料一次過整批送回記憶體
write back快很多，早期的 cache 只有 write through 模式，但現在的 cache 都使用 write back 模式

45

pipline越多越好還是越少越好？多的缺點是什麼？

pipeline分得剛剛好最好，分得越細越完善，越可以使得指令完成的總速度提高
多的缺點：級數無法被無窮切割，此外級數的提高也會導致數據和指令衝突的嚴重性提高，反而會降低效率。

46

[c] static 用法？

C語言static有兩個目的
static storage （靜態存儲）：變數儲存空間為static storage而非function stack，使變數生命週期與程式
internal linkage（內部連結）：限制變數可視範圍於檔案內。（聽說這是後來加的，為了減少保留字的使用所以
function使用static修飾
一個 static函式表示，其可以呼叫的範圍限於該原始碼文件之中，如果有些函式僅想在該原始程式文件之中使用，

47

Cache coherence

在計算機科學中，快取一致性（英語：Cache coherence，或cache coherency），又譯為快取連貫性、快取同調
在一個系統中，當許多不同的裝置共享一個共同記憶體資源，在快取記憶體中的資料不一致，就會產生問題。這個問
快取一致性可以分為三個層級：
在進行每個寫入運算時都立刻採取措施保證資料一致性
每個獨立的運算，假如它造成資料值的改變，所有行程都可以看到一致的改變結果
在每次運算之後，不同的行程可能會看到不同的值（這也就是沒有一致性的行為）

Directory-based Protocol
首先要來了解一下這個 protocol 用三種狀態來描述Data的狀況
分別用U,S,E 這三個代號來簡稱
Uncached ：data 的狀態目前是閒置的(沒有正在被share)所以可以放心拿去用
Shared：data 的狀態目前是被shared，也就是有其他的CPU把data copy去他的cache裡面使用，此時還是可以把
Exclusive：這是mutual exclusive的意思，表示現在data的狀態正在被修改中，其他人不可以使用！有再用的人

快取一致性的問題
這裡用snooping維持一致性
採用wrirte-invalidate和write back
write-invalidate簡單說是write miss時
他會送出無效訊號把其它拷貝毒死再更新自己的資料

每個processor會有一個snoop tag打聽bus上的資訊
而快取區塊會有三種狀態shared,exclusive,invalid
當read miss、write miss狀態就會轉換
而read hit不會。

以下(1)~(6)為題目的路徑標示
(1)invalid→shared
當此無效區塊發生read miss，則直接把要讀的block搬上來變shared
(2)shared→shared
當此共享區塊發生read miss，因為本來區塊就是乾淨的所以還是可以共享
(3)shared→exclusive
當此共享區塊發生write miss，則送出Invalidate訊息把所有copy殺掉
再讀要的區塊並寫入，改成exclusive
(4)exclusive→shared
當此互斥區塊發生read miss，因為髒髒被寫過而且別人沒有此區塊資料所以
要write back回memory再變成shared
(5)exclusive→exclusive
當此互斥區塊發生write miss，一樣要先write-back回memory但他還是髒的。
(6)invalid→exclusive
當此無效區塊發生write miss，則讀取所需區塊並寫入，然後改成互斥狀態。

48

cache line

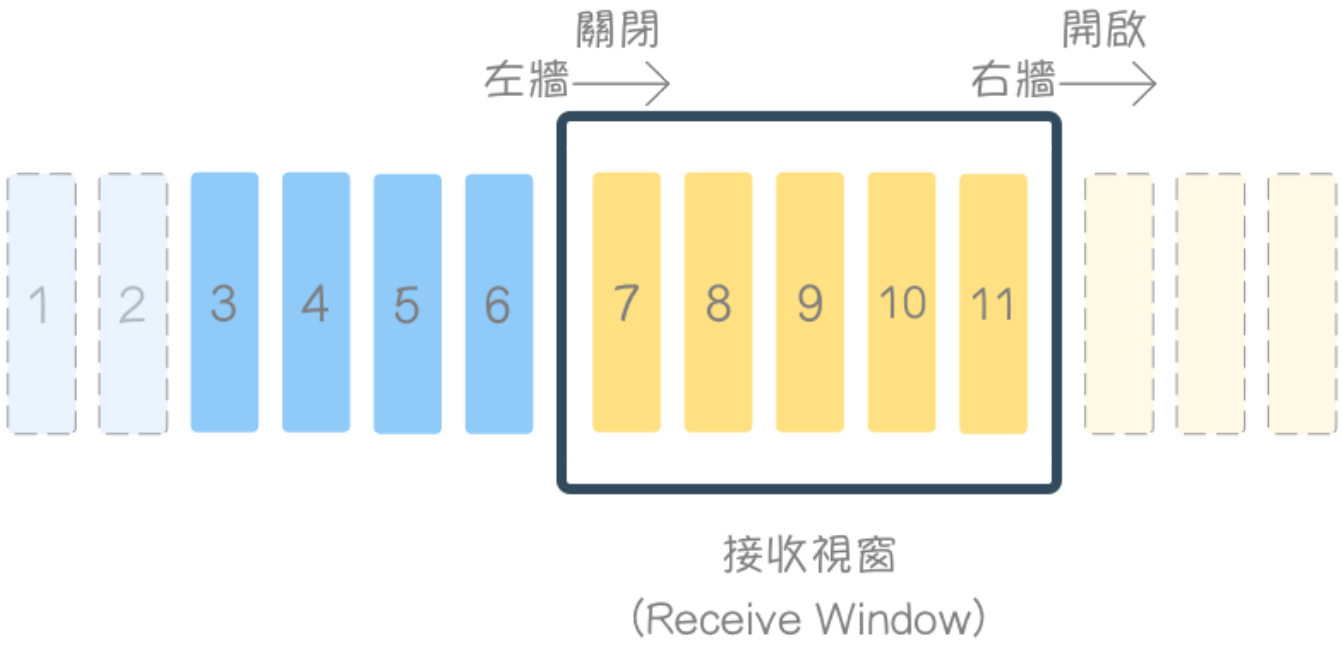
cache 存取資料時，通常是分成很多小單位，稱為 cache line。
例如，Pentium III 的 cache line 長度是 32 bytes。
也就是說，如果 CPU 要讀取記憶體位址 0x00123456 的一個 32 bits word（即 4 bytes），且 cache 中沒有
所以，當 CPU 讀取連續的記憶體位址時，資料都已經讀到 cache 中了。

49

tcp flow control

用於平衡 傳送端 與 接收端的流量，
避免 高速傳送端 癱瘓了 低速接收端。

傳輸控制協定 (Transmission Control Protocol, TCP) ，
由於 發送者 (Sender) 與 接收者 (Recipient) 傳輸、讀取的速率不相等，
接收端 會將資料暫存在 接收緩衝區 (Receiving Buffers)，
並等待 應用層讀取後 (消化)，再從 接收緩衝區 中清除。



接收緩衝區 (Receiving Buffers)，實際上就是一段記憶體位址，
時常使用 環狀佇列 (circular queue) 資料結構來實作，
使空間有效率的復用，並避免大量資料的搬移。
1、2 ，為應用層已讀取，
並從 接收緩衝區 清除之位元組。
3、4、5、6 為已接收 並 確認 (acknowledged) 收到，
應用層尚未讀取 (消化) 之位元組。
在 接收視窗 內的 7、8、9、10、11 為等待接收資料的 空的緩衝區，
3、4、5、6、7、8、9、10、11 合計就是 接收緩衝區大小 ！
超出 接收視窗大小 (Receive Window Size, rwnd) 的部分則無法接收。

當 接收端 收到資料，並回覆 確認 (acknowledgment) 收到，
接收視窗 會 關閉 (Close) (左牆 往右移)。

當 接收段 應用層讀取資料 (消化) 時，資料會被清除，
接收視窗 會 開啟 (Open) (右牆 往右移)。

TCP 透過 錯誤控制 (Error Control) 機制，
確保了傳輸的可靠性 (reliable)，
其中最常見的是: 檢驗和 (checksum)、確認 (acknowledgment)、重送 (retransmission)。
TCP 會為每個連線啟動 一個 逾時重送 (Retransmission Time-out, RTO) 計時器 ⏰，
當時間到期，尚未收到接收端的 確認 (acknowledgment) 回覆，
則視同區段毀損 (遺失、延遲)，並 重送 該 區段 (Segment)，
這便是為何 發送完的資料，會等到接收者回覆確認後，才從佇列中清除。

TCP 流量控制 (TCP Flow Control)，
用於平衡 傳送端 與 接收端的流量，
避免 高速傳送端 癱瘓了 低速接收端。
也就是透過 回饋 (feedback):

TCP 的 發送視窗大小，主要由 接收端 來維護，
每次回報確認時，皆會告知傳送端 接收視窗 (rwnd) 大小。

這種 動態調整視窗大小，來實現流量控制的方式，
即是鼎鼎大名的 — — 滑動視窗 (Sliding Window)。
如果滿了的 接收緩衝區，傳送端 繼續傳送? 😉
— — 接收端 丟棄區段

50

Congestion Control

慢啟動 (Slow Start)、壅塞避免 (Congestion Avoidance)、
快速重送 (Fast Retransmit) 和 快速恢復 (Fast Recovery)

為每個 TCP 連接，新增的兩個狀態變數:
壅塞視窗 (Congestion Window, cwnd)
慢啟動門檻 (Slow Start Threshold, ssthresh)

判別 當區段遺失 時，接收端 採取的 重送機制
– – 逾時重送 (Retransmission Timeout)、快速重送 (Fast Retransmission)。
這是因為:
在網際網路平穩運行時，TCP 傳輸錯誤的比例很低，
當區段遺失時，TCP 假設遺失是 壅塞 所造成。
當 接收端 使用 逾時重送 (Retransmission Timeout):
接收端 在 逾時之前，未收到 3 個 區段 或 送回的 確認 (ACK) 遺失 – – 壅塞嚴重 的徵兆。
當 接收端 使用 快速重送 (Fast Retransmit):
接收端 在 逾時之前，已收到 3 個 區段 – – 壅塞輕微 的徵兆。

最大區段長度 (Maximum Segment Size, MSS)，雙方會在連線建立時決定，
且這條連線運作期間街不會改變，若其中一方未定義，則使用預設值: 536 bytes。
可別被他的名字唬了，它定義的是所能接收 『資料』的最大長度，而非區段!
(即不包含 TCP 表頭 + 選項)

最大傳輸單元 (Maximum Transmission Unit, MTU)
不同的 資料鏈結/實體 層，擁有不同的 MTU 大小，
最常見的 為 乙太網路 (Ethernet) 的 1500 個位元組。
TCP 透過 路徑 MTU 探索 (Path MTU Discovery)、其他演算法 或 經驗法則，
調整 最大區段長度 (Maximum Segment Size, MSS)。

壅塞視窗 (Congestion Window, cwnd)
在接收到 確認 (ACK) 區段前，能夠傳輸的最大資料總量。
如上所述，cwnd 「概念上」時常以 MSS 做為自然單位。

慢啟動 (Slow Start)
TCP 開始傳輸到未知條件的網絡時，
需緩慢地探測網絡，以確定可用容量，
避免因不適當的大量數據並發而使網絡壅塞。
TCP 傳送端 使用 慢啟動 與 壅塞避免，
控制注入網路的未完成資料量。

慢啟動 不如其名，一點都不「慢」，
相反地，其 cwnd 增長速度非常快速，為 指數成長 (exponential growth)。
連線建立後，cwnd 大小為 初始視窗 (Initial Window, IW)，
TCP 進入 慢啟動階段:
    TCP 每次接收 並 確認 (ACK) 新資料時，
    增加 至多 1 * SMSS 位元組 的 壅塞視窗 (cwnd) 大小。

慢啟動 不能無止盡的增加 壅塞視窗 (cwnd) 大小，
否則失去了壅塞「控制」的意義。
慢啟動門檻 (ssthresh)，為 TCP 另一狀態變數，
或稱 慢啟動門閥、門檻值、閾 (ㄩˋ) 值，
不要說 閥 (ㄈㄚˊ) 值 就對了 😂。
當 cwnd 到達此門檻，或 觀察到壅塞時，
即停止 指數成長 的 慢啟動，進入 線性成長 的 壅塞避免。

壅塞避免 (Congestion Avoidance)
壅塞避免 期間，每個 往返時間 (RTT)，
大約增加 1 SMSS 大小 的 壅塞視窗 (cwnd)，
且不應超過，儘管某些規範嘗試如此 (e.g., [RFC 3465])，
壅塞避免 持續到 偵測到壅塞為止。

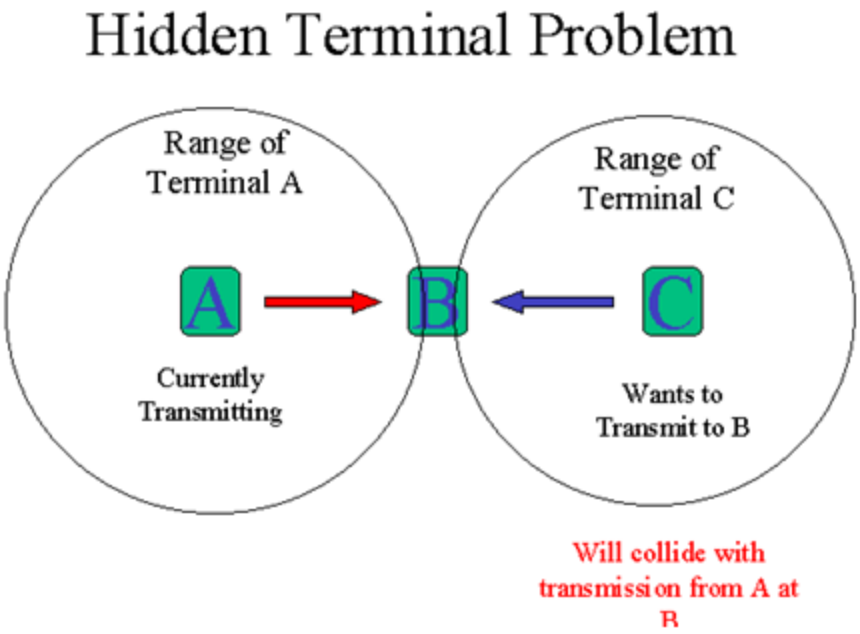
快速重送 (Fast Retransmit)
接收端:
  某區段遺失，代表後續接收的區段 順序錯誤，
  因此，不應使用 延遲確認 (Delayed ACK)，
  而是立即確認 (immediate ACK)。
目的是為了讓 傳送端 知道:
接收到的是 亂序 (out of order) 的區段
預期接收的序列號

傳送端:
  當 接收到 3 個 重複的確認 (duplicate ACK)，
  則立即重送該區段，而非等到重送計時器逾時。

快速恢復 (Fast Recovery)
當 快速重送 送出「似乎是」遺失的區段後，
TCP 進入 快速恢復 演算法階段，
直到收到 非重複 的 確認 (ACK) 區段。
不執行 慢啟動 的原因是:
重複的確認區段，除了是區段遺失的跡象，
往往也代表 該區段 已離開網路，
因此，我們得知它已不再消耗網路資源。

51

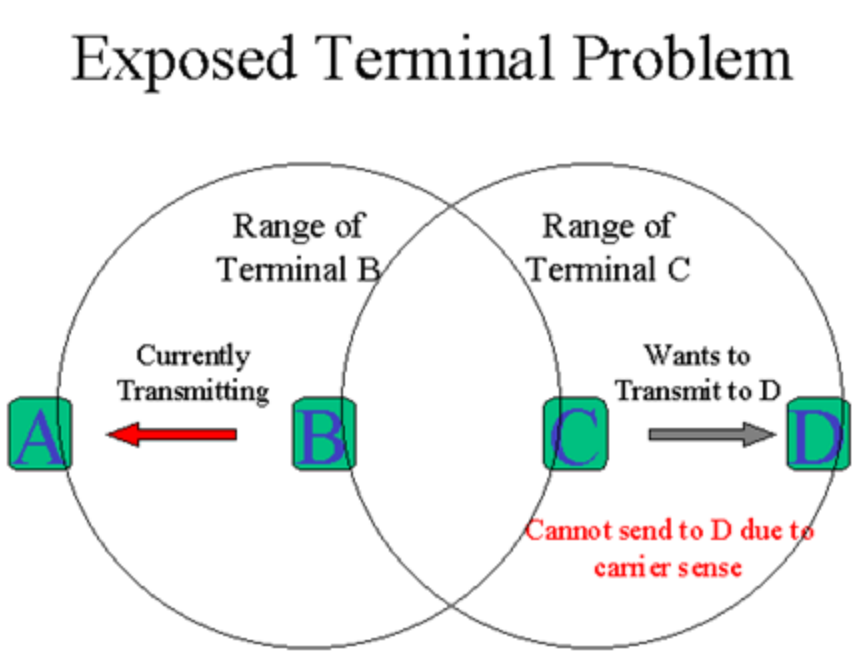
Hidden Terminal Problem



隱藏節點問題:
節點B在節點A和節點C傳輸範圍內的交集區域內，但是A和C都不在互相的傳輸範圍內，這時同時有兩個節點A,C想傳送

這種因傳送距離而發生的誤判的問題稱為隱藏節點問題(Hidden Terminal Problem).

解決辦法:
RTS/CTS
此機制解決 hidden terminal problem，如圖中Station A 與Station C同時傳送封包給B，當Station A要送封



暴露節點問題
C 要傳送資料給D時，發現(聽到)傳輸範圍內的B正在傳送資料給A（C是B的暴露節點），C就會延遲傳送（但這種延遲

解決辦法:
RTS/CTS
802.11 RTS / CTS的機制，有助於解決這個問題只如果節點是同步的，
數據包大小和數據速率是相同的兩個節點的傳輸。
當一個節點聽到一個RTS從鄰近的節點，而不是相應的CTS，
該節點可以推斷，這是一個裸露的節點，是允許傳輸給其他鄰近的節點。

52

CSMA/CD, CSMA/CA

首先我們必須要有一個體認，在同一個網路媒體上，同一時間只能允許一個訊號在傳送。如果同一媒體在同一時間中

一般來說乙太網路 (Ethernet) 的標準 IEEE 802.3 使用的就是 CSMA/CD (Carrier Sense Multiple Access
然而無線乙太網路 (Wireless Ethernet) 的標準 IEEE 802.11 使用的為 CSMA/CA (Carrier Sense Multipl
從字面上來看差異在 CD 與 CA 不同，一個是碰撞偵測處理 Collision Detection，另一個則是避免碰撞 Colli

CSMA/CD（載波偵測多重存取/碰撞偵測）運作過程如下：
- 訊號採用廣播的方式傳送（所以才會發生碰撞）
- 當節點要發送訊號時，會先偵測通道是否有其他節點正在使用（carrier sense）
- 當通道沒有被其他節點使用時，就傳送封包
- 封包傳送之後立即檢查是否發生碰撞（carrier detection），若是發生碰撞則對通道發出高頻訊號高知其他節點
- 碰撞後隨機等待一段時間重新發送封包
- 嘗試 15 次都失敗的話則告知上層 Timeout

CSMA/CA（載波偵測多重存取/碰撞避免）運作過程如下：
訊號採用廣播的方式傳送（非常容易受到無線電波干擾）
當節點要發送訊號時偵測頻道是否空閒
若是空閒則等待 IFS, Interval Frame Space 時間後再次偵測頻道是否空閒
若是空閒則發送封包，反之重新進入等待頻道空閒（隨機等待時間）
發送 RTS 之後必須在限定時間內收到來至目的端的 CTS 訊號
(Request To Send/Clear To Send)
當失敗 32 次之後通知上層 Timeout

53

malloc free

一維陣列
這是使用 malloc 與 free 配置一維動態陣列的例子。

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  // 用來管理動態記憶體的指標
  int *dynArr;

  // 指定空間大小
  int arrLen = 10;

  // 取得記憶體空間
  dynArr = malloc( arrLen * sizeof(int) );

  if( dynArr == NULL ) {
    // 無法取得記憶體空間
    fprintf(stderr, "Error: unable to allocate required memory\n");
    return 1;
  }

  // 使用動態取得的記憶體空間
  int i;
  for (i = 0; i < arrLen; ++i) {
    dynArr[i] = i;
    printf("%d ", dynArr[i]);
  }

  // 釋放記憶體空間
  free(dynArr);

  return 0;
}
```

在 C 語言中動態配置的記憶體都必須配合指標來管理，
這個範例中我們需要動態建立一個整數（int）的陣列，
所以一開始先宣告一個整數的指標 dynArr，
接著使用 malloc 配置指定大小的記憶體空間給這個陣列使用。

malloc 在配置新的記憶體空間之後，
會傳回指向該空間第一個位元組（byte）的指標，
而在無法取得新的記憶體空間時（例如系統的記憶體不夠的時候），
就會傳回 NULL，所以在使用動態配置的記憶體空間之前，
要先檢查是否有配置成功。

當記憶體空間使用完之後，
再呼叫 free 來將該記憶體空間釋放掉，
這個動作不可以忘記，
否則就會造成記憶體洩漏（memory leak）的問題。

一般來說在程式中只要呼叫一次 malloc，
後續就要對應一次的 free 呼叫，
確保每一次配置的記憶體在使用完之後，
都有被妥善釋放。

二維陣列
這是拿一塊動態配置的記憶體空間，建立二維陣列的一種作法：

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *dynArr;

  // 指定空間大小
  int arrLen1 = 10;
  int arrLen2 = 5;

  // 取得記憶體空間
  dynArr = malloc( arrLen1 * arrLen2 * sizeof(int) );

  if( dynArr == NULL ) {
    fprintf(stderr, "Error: unable to allocate required memory\n");
    return 1;
  }

  // 使用動態取得的記憶體空間
  int i, j;
  for (i = 0; i < arrLen1; ++i) {
    for (j = 0; j < arrLen2; ++j) {
      int index = i * arrLen2 + j;
      dynArr[index] = index;
      printf("%d ", dynArr[index]);
    }
  }

  // 釋放記憶體空間
  free(dynArr);

  return 0;
}
```

calloc 函數
除了使用 malloc 之外，
也可以使用 calloc 配置初始化的陣列記憶體，
這兩個函數用法大同小異，
只是參數的指定方式不同，以及有無初始化而已，
透過 calloc 所取得的記憶體空間，
其值會被自動初始化為 0 或 NULL，
在釋放記憶體時跟 malloc 一樣都是呼叫free 函數並傳入指向記憶體的指標

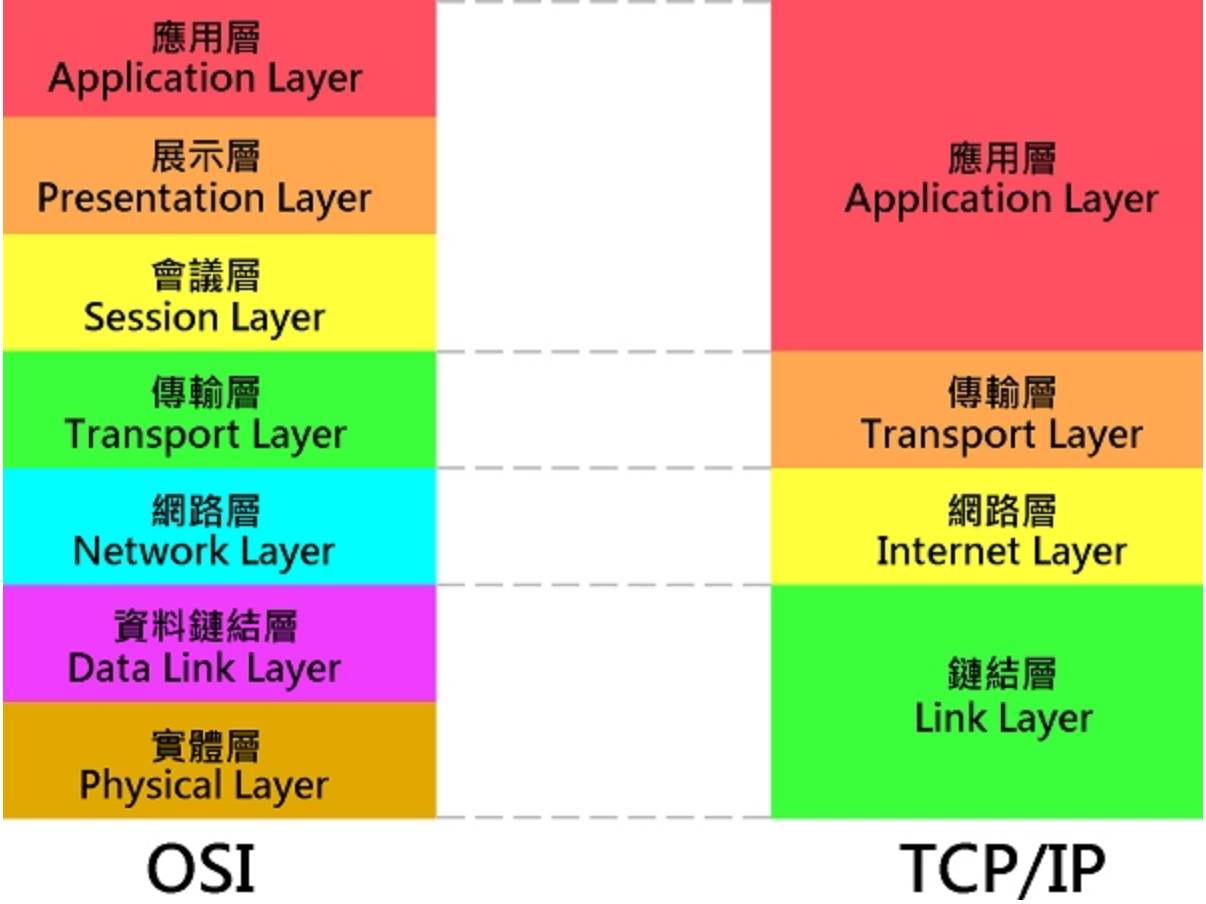
memset 函數
另外一種初始化的方式是使用 memset 函數：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
  int *dynArr;
  int arrLen = 10;
  // 配置未初始化的記憶體空間
  dynArr = malloc( arrLen * sizeof(int) );
  if( dynArr == NULL ) {
    fprintf(stderr, "Error: unable to allocate required memory\n");
    return 1;
  }
  // 將記憶體初始化為 0
  memset(dynArr, 0, arrLen * sizeof(int));
  int i;
  for (i = 0; i < arrLen; ++i) {
    printf("%d ", dynArr[i]);
  }
  free(dynArr);
  return 0;
}
```

memset 函數的第一個參數是指向記憶體的指標，
第二個參數是一個常數值，
在初始化時會將此常數轉換為 unsigned char 型別，
指定給記憶體中的每一個位元組，
而第三個參數則是記憶體空間的大小。

54

TCP/IP Model



55

如何實現 Mutex

Peterson's Algorithm

使用兩個控制變數 `flag` 與 `turn` 。其中 `flag[n]` 的值為 `true` ，表示 ID 為 `n` 的 Process 希望進入該臨界區。變數 `turn` 保存有權訪問共享資源的 Process ID.
該演算法滿足解決臨界區問題的三個必須標準：互斥訪問、進入(No Deadlock)、有限等待(No Starvation)。

```
//flag[] is boolean array; and turn is an integer
flag[0]   = false;
flag[1]   = false;
int turn;
```

```
P0: flag[0] = true;
    turn = 1;
    while (flag[1] == true && turn == 1)
    {
        // busy wait
    }
    // critical section
    ...
    // end of critical section
    flag[0] = false;



P1: flag[1] = true;
    turn = 0;
    while (flag[0] == true && turn == 0)
    {
        // busy wait
    }
    // critical section
    ...
    // end of critical section
    flag[1] = false;
```

(Reference: https://www.wikiwand.com/en/Peterson's_algorithm (https://www.wikiwand.com/en/Peterson%27s_algorithm))