1. OS：

1.1. 何謂reentrant程式，設計reentrant需注意什麼？

執行過程中如果被中斷後而且在之前的調用完成之前，可以再安全地呼叫一次，即可稱作reentrant。

不可使用static, global等變數。

程式本身不能修改自己的code。

不要call任何違反上述規則的function。

在interrupt context裡執行的必須為reentrant function。

會被多個threads/tasks call的function必須為reentrant。

1.2. 解釋stack與heap

Stack 是拿來給程式呼叫function時存放function資料用的，而Heap是用來存放並且管理，程式全部所需要用到的變數與資料。

1.3. 何謂deadlock?

一組processes陷入互相等待的情況(Circular waiting)

造成processes皆無法往下執行，使得CPU utilization及Throughput大幅降低

條件：no preemption, hold and wait, mutual exclusion, circular waiting

1.4. 說明 mutex 與 semaphore

- 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：一個是只能由持鎖人解鎖、一個是在釋放鎖之前不能退出的特性，讓 Mutex 較常使用在 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。

1.5. 設計OS的重點在哪些？

排程、記憶體管理、檔案系統、網路、安全、使用者介面、驅動程式

1.6. 如何 Linux 與 windows 互相傳送檔案？

Scp，需安裝ssh for windows。

1.7. 何謂DLL?

DLL 是一種包含可供多個程式同時使用的程式碼和資料的文件庫。

當執行檔呼叫到DLL檔內的函式時，Windows作業系統才會把DLL檔載入記憶體內，DLL檔本身的結構就是可執行檔，當程式有需求時函式才進行連結。透過動態連結方式，記憶體浪費的情形將可大幅降低。靜態連結函式庫則是直接連結到執行檔。

1.8. uClinux 與 Linux 最大差異在哪？

A. 缺乏記憶體管理，沒有virtual memory，不支援page
B. 不能執行時增加thread
C. 可執行程式不是elf， 而是flat
D. 不能用fork， 而是用vfork
E. RAMDISK

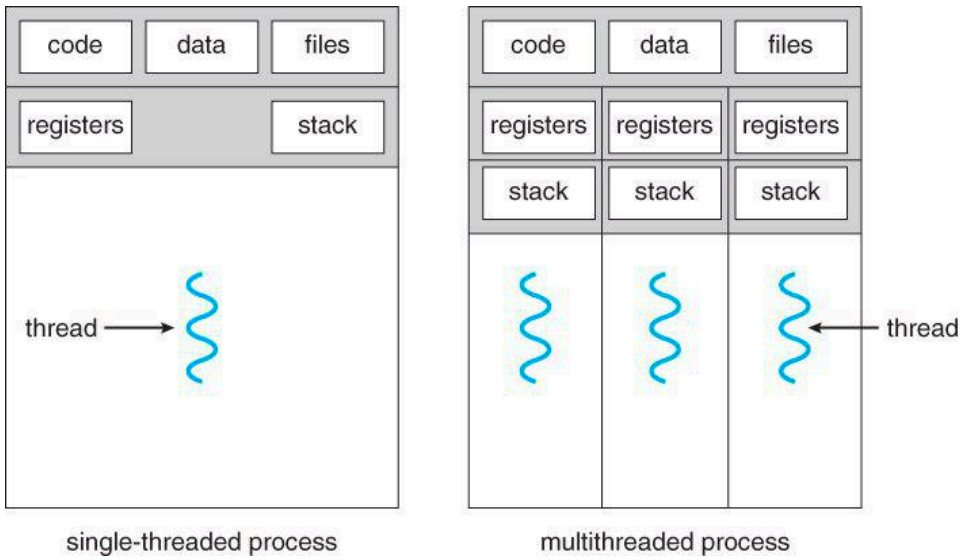
1.9. 何謂即時多工系統？

如果有一個任務需要執行，即時作業系統會馬上（在較短時間內）執行該任務，
Hard RTOS必須使任務在確定的時間內完成。
Soft RTOS能讓絕大多數任務在確定時間內完成。
特徵：多任務、有線程優先級、多種中斷級別。

1.10. Process & Thread

A. Process
- Process 是電腦中已執行 Program 的實體。
- 每一個 Process 是互相獨立的。
- Process 本身不是基本執行單位，而是Thread (執行緒)的容器。
- Process 需要一些資源才能完成工作，如 CPU、記憶體、檔案以及I/O裝置。
- 在多功作業系統(Multitasking Operating System)中，可以同時執行數個Process ，然而一個 CPU 一次只能執行一個 Process (因此才有現在的多核處理器)，而 Process 的運行量總量不會少於 CPU 的總量，又Process 會佔用記憶體，因此如何排程(Scheduling，恐龍本第五章) 、如何有效管理記憶體(恐龍本第八章)則是 OS 所關注的事。

B. Thread
- 同一個 Process 會同時存在多個 Thread。
- 同一個 Process 底下的 Thread 共享資源，如 記憶體、變數等，不同的Process則否。
- 在多執行緒中(Multithreading)，兩個執行緒若同時存取或改變全域變數(Global Variable)，則可能發生同步(Synchronization，恐龍本第六章)問題。若執行緒之間互搶資源，則可能產生死結(Deadlock，恐龍本第七章)，同樣的，如何避免或預防上述兩種情況的發生，依然是 OS 所關注並改善的。



2. 計組、硬體：

2.1.何謂DMA，有何好處？

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CPU介入處理。

Advantages:

A. Transferring the data without the involvement of the processor will speed up the read-write task.
B. DMA reduces the clock cycle requires to read or write a block of data.
C. Implementing DMA also reduces the overhead of the processor.

cache coherence problem

2.2.何謂Little endian / Big endian

Little endian：最低位元組在最低位址

Big endian：最高位元組在最低位址

2.3.何謂 JTAG? 何謂ICE？

JTAG：聯合測試工作群組。此標準用於驗證設計與測試生產出的印刷電路板功能。

ICE：

2.4.解釋 write back 與 write through

- write through：CPU向cache寫入數據時，同時向memory(後端存儲)也寫一份，使cache和memory的數據保持一致。優點是簡單，缺點是每次都要訪問memory，速度比較慢。

- post write：CPU更新cache數據時，把更新的數據寫入到一個更新緩沖器，在合適的時候才對memory(後端存儲)進行更新。這樣可以提高cache訪問速度，但是，在數據連續被更新兩次以上的時候，緩沖區將不夠使用，被迫同時更新memory(後端存儲)。

- write back：cpu更新cache時，只是把更新的cache區標記一下，並不同步更新memory(後端存儲)。只是在cache區要被新進入的數據取代時，才更新memory(後端存儲)。這樣做的原因是考慮到很多時候cache存入的是中間結果，沒有必要同步更新memory(後端存儲)。優點是CPU執行的效率提高，缺點是實現起來技術比較復雜。

- write-miss policies:
  - Write allocate (also called fetch on write): data at the missed-write location is loaded to cache, followed by a write-hit operation. In this approach, write misses are similar to read misses.
  - No-write allocate (also called write-no-allocate or write around): data at the missed-write location is not loaded to cache, and is written directly to the backing store. In this approach, data is loaded into the cache on read misses only.
- A write-back cache uses write allocate, hoping for subsequent writes (or even reads) to the same location, which is now cached.
- A write-through cache uses no-write allocate. Here, subsequent writes have no advantage, since they still need to be written directly to the backing store.

2.5.列舉幾個serial port, parallel port

Sr. No.	Key	Serial Ports	Parallel Ports
1	Purpose	Serial Port is used for serial data transmission.	Parallel Port is used for parallel data transmission.
2	Transmission Speed	Transmission speed of a serial port is slow as compared to a parallel port.	Transmission speed of a parallel port is quiet high as compared to a serial port.
3	Redundancy	Bottom-Up model is better suited as it ensures minimum data redundancy and focus is on re-usability.	Top-down model has high ratio of redundancy as the size of project increases.
4	No. Of Wires	Wire connections to serial port are quiet less as compared to parallel port.	No. of wires that are connected to parallel port are quiet high as compared to serial port.
5	Capability	A serial port is able to transmit a single stream of data at a time.	A parallel port is able to transmit multiple data streams at a time.
6	Data Sending Mechanism	A serial port sends data bit by bit after sending a bit at a time.	A parallel port sends data by sending multiple bits in parallel fashion.
7	Port Type	A serial port uses Male ports.	A parallel port uses Female ports.
8	Applications	Modems, security cameras, device controllers use serial ports.	Printers, Hard Drives, CD drives use parallel ports.

2.6.說明 Watchdog 之運作機制

當系統的主程式發生某些錯誤事件時，如假死機或未定時的清除看門狗計時器的內含計時值（多半是向對計時器發送清除訊號），這時看門狗計時器就會對系統發出重設、重新啟動或關閉的訊號，使系統從懸停狀態回復到正常運作狀態。

2.7.Hazard

<table>
<tr><td colspan="7">Hazard solution</td></tr>
<tr><td>Structural</td><td colspan="4">Data</td><td colspan="2">Control</td></tr>
<tr><td rowspan="2">Add Hardware、stall</td><td colspan="2">Software (Compiler)</td><td colspan="2">Hardware</td><td colspan="2">Software (Compiler)</td><td rowspan="2">Hardware (Dynamic branch prediction)</td></tr>
<tr><td>Insert nops</td><td>Instruction Reordering</td><td>Forwarding</td><td>Stall + Forwarding</td><td>Insert nops</td><td>Insert safety Instruction (Delay Branch)
1. Predict not taken
2. Flush wrong instruction</td></tr>
</table>

	Hardware Based						Software based			
	Stall	Add HW	FW	Stall+FW	Branch Pred.	MV B decision earlier	Insert NOP	Instr. Reorder	Delayed branch	Pred.
Structure	O	O								
Data	O		O	O			O	O		
Control	O				O	O	O		O	O

3. 網路：

3.1. 分別說明switch、hub、router、gateway

A. Hub(集線器)

在星狀拓樸網路(star topology)中，扮演連接或重新建立訊號的角色，可擴大類比或者是數位訊號。在區域網路(LAN)中，電腦與電腦利用網路連接時，如果用Hub連接，即使有任何一段線路出問題，只會有一台電腦無法運作，不會影響網路中其他電腦的作業。

在接收封包(Packet)進來之後，會將這個封包送到其它所有的電腦(即廣播，每一台電腦都會收到該封包)，不管誰才是應該收到該封包的電腦。

B. Switch(交換器)

Switch 的作用是在區域網路中，將網路作連接的動作。Switch 有一個table，記錄著每一台電腦的MAC 位址，當封包進來之後，Switch 會去檢查該封包的目的地的是哪一個MAC 位址的電腦，只將這個封包送給該台電腦，其他電腦則不會收到封包。

Switch的傳輸方式

(1). Cut Through：接收到目的地址後即轉發出去。延時小，但壞的資料一樣轉發。

(2). Store-and-Forward：接收到完整的資料包後，校驗好壞，好的轉發，壞的丟棄重發。傳輸可靠，但延時較長。

(3). Fragment free：接收到資料包後，大於64bytes的轉發，小於64bytes的丟棄。好壞介於上述兩種方式之間。

C. Router(路由器)

路由器是用來將網路的資訊, 使用在電腦之間傳送的基本設備, 路由器的工作在於OSI 模式第三層(網路層)，用來決定資料傳遞路徑的設備. 我們使用的IP協定就是藉由路由器將不同的IP位址連接在一起。網路上的資料分成一段一段的封包packet，而這些封包要指向何處便是由路由器來決定的，路由器會根據資料的目的地，指示正確的方向，計算評估最便捷有效率的路徑來傳輸資料，也就是說路由器要為封包做最佳化的工作，找出最適當的路徑. 路由器通常最少會有兩個介面，而這兩個介面分別區隔不同的IP網段。例如IP分享器有WAN和LAN兩種介面，區隔WAN的實際IP與LAN的虛擬IP網段。

Router 與Bridge 的另一個不同點在於：

Bridge 只是單純做為一個決定是否讓封包通過的橋樑，Router 則會執行選徑功能(OSPF,EIGRP,...etc.)。

D. Gateway

A gateway, as the name suggests, is a passage to connect two networks together that may work upon different networking models. They basically work as the messenger agents that take data from one system, interpret it, and transfer it to another system. Gateways are also called protocol converters and can operate at any network layer. Gateways are generally more complex than switches or routers. Gateway is also called a protocol converter.

3.2. 何謂IP fragment?

網際網路協定（IP）允許資料包（英語：Datagram）進行分片（英語：fragmentation，又譯分段），將資料包分割成更小的單位。這樣的話，當封包比鏈路最大傳輸單元（MTU）大時，就可以被分解為很多的足夠小片段，以便能夠在其上進行傳輸。

3.3.說明DHCP server功能

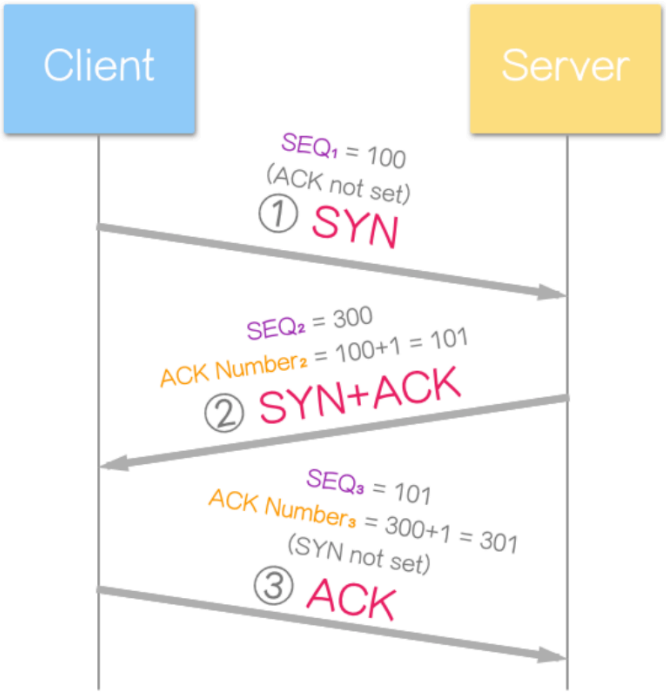
用於內部網路或網路服務供應商自動分配IP位址給使用者

用於內部網路管理員對所有電腦作中央管理

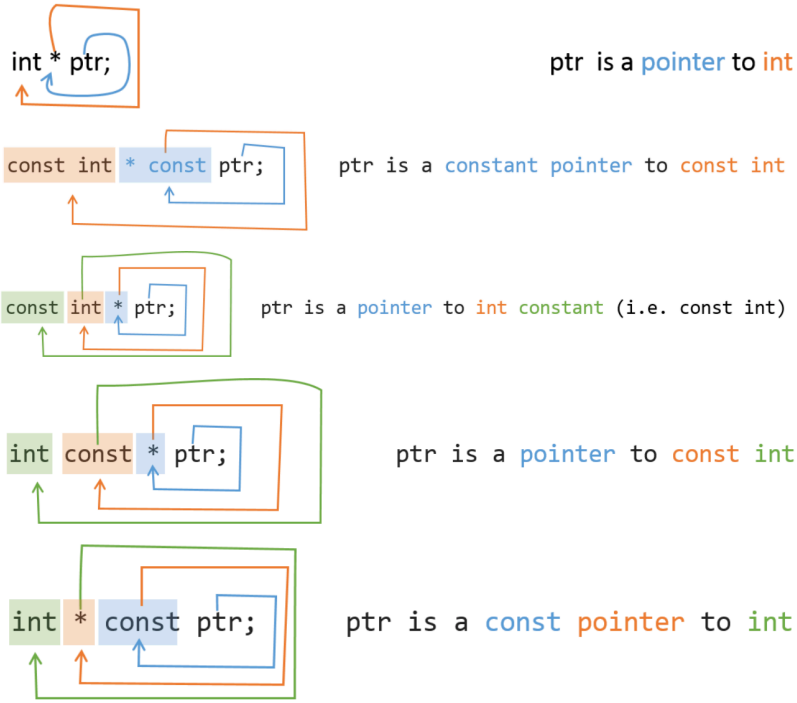
3.4. 說明IP、subnet mask

subnet mask：它是一種用來指明一個IP位址的哪些位標識的是主機所在的網路位址以及哪些位標識的是主機位址的位遮罩。

3.5.說明 3-way handshaking



4. C/C++：



[用心去感覺] const vs #define

1. 編譯器處理方式 : define 在預處理階段展開；const 在編譯階段使用。
2. 類型和安全檢查 : const 會在編譯階段會執行類型檢查，define 則不會。
3. 存儲方式 : define 直接展開不會分配記憶體，const 則會在記憶體中分配。

六、關鍵字 inline

inline 可以將修飾的函式設為行內函式，即像巨集 (define) 一樣將該函式展開編譯，用來加速執行速度。

inline 和 #define 的差別在於：

- inline 函數只對參數進行一次計算，避免了部分巨集易產生的錯誤。
- inline 函數的參數類型被檢查，並進行必要的型態轉換。
- 巨集定義盡量不使用於複雜的函數
- 用 inline 後編譯器不一定會實作，僅為建議。

[用心去感覺] const 和 volatile 合用

```
extern const volatile unsigned int rt_clock;
```

這是在 RTOS kernel 常見的一種宣告：rt_clock 通常是指系統時鐘，它經常被時鐘中斷進行更新。所以它是 volatile。因此在用的時候，要讓編譯器每次從記憶體裡面取值。而 rt_clock 通常只有一個寫者（時鐘中斷），其他地方對其的使用通常都是唯讀的。所以將其聲明為 const，表示這裏不應該修改這個變數。所以 volatile 和 const 是兩個不矛盾的東西，並且一個物件同時具備這兩種屬性也是有實際意義的。

2. 引入防護和條件編譯

引入防護 (Include guard) 是一種條件編譯，用於防範 #include 指令重複引入的問題。

```
/* 避免重複引入 */
#ifndef MYHEADER
#define MYHEADER
...
#endif
```

第一次被引入時會定義巨集 MYHEADER，再次引入時判斷 #ifndef 測試失敗，因此編譯器會直接跳到 #endif，由此避免了重複引用。另有非標準的指令 #pragma once 提供相同效果，但由於可攜性不如上例，因此大多時候還是上面提到的方法為主。

條件編譯還有一些其它應用：

```
/* 若前處理器已經 define MYHEADER，就編譯 part A，否則編譯 part B。 */
#ifdef MYHEADER
#define MYHEADER
   // part A
#else
   // part B
#endif

/* DEBUG flag */
#ifdef DEBUG
   print ("device_open(%p) ", file);
#endif
```

```
//對指標不熟悉的使用者會以為以下的程式碼是符合預期的
//錯誤例子：
void newArray(int* local, int size)
{
    local = (int*) malloc( size * sizeof(int) );
}
int main()
{
    int* ptr;
    newArray(ptr, 10);
}
//接著就會找了很久的 bug，最後仍然搞不懂為什麼 ptr 沒有指向剛剛拿到的合法空間
//讓我們再回顧一次，並且用圖表示
//原因如下：
//    1. int* ptr;                      ptr -> |__未知的空間__|
//    2. 呼叫函式 newArray              ptr -> |__未知的空間__| <- local
//    3. malloc 取得合法空間           ptr -> |__未知的空間__|
//                                             |___合法空間___| <- local
//    4. 離開函式                       ptr -> |__未知的空間__|
//用圖看應該一切就都明白了，我也不需冗言解釋
//也許有人會想問，指標不是傳址嗎？
//精確來講，指標也是傳值，只不過該值是一個位址 (ex: 0xfefefefe)
//local接到了ptr指向的那個位置，接著函式內local要到了新的位置但是ptr指向的位置還是沒變的，因此離開函式後就好像事什麼都沒發生，嚴格說起來還發生了 memory leak。
```

```
//以下是一種解決辦法：
int* createNewArray(int size) {
    return (int*) malloc( size * sizeof(int) );
}
int main() {
    int* ptr;
    ptr = createNewArray(10);
}

//改成這樣亦可   ( 為何用 int** 就可以？想想他會傳什麼過去給local )
void createNewArray(int** local, int size) {
    *local = (int*) malloc( size * sizeof(int) );
}
int main() {
    int *ptr;
    createNewArray(&ptr, 10);
}

//如果是 C++可用 Reference：
void newArray(int*& local, int size) {
   local = new int[size];
}
```

中斷（Interrupts）

11. 中斷是嵌入式系統中重要的組成部分，這導致了很多編譯開發商提供一種擴展—讓標準C支持中斷。具代表事實是，產生了一個新的關鍵字 __interrupt。下面的代碼就使用了__interrupt關鍵字去定義了一個中斷服務子程序(ISR)，請評論一下這段代碼的。

```
__interrupt double compute_area (double radius)
{
    double area = PI * radius * radius;
    printf("\nArea = %f", area);
    return area;
}
```

這個函數有太多的錯誤了，以至讓人不知從何說起了：

1)ISR 不能返回一個值。如果你不懂這個，那麼你不會被僱用的。

2) ISR 不能傳遞參數。如果你沒有看到這一點，你被僱用的機會等同第一項。

3) 在許多的處理器/編譯器中，浮點一般都是不可重入的。有些處理器/編譯器需要讓額處的寄存器入棧，有些處理器/編譯器就是不允許在ISR中做浮點運算。此外，ISR應該是短而有效率的，在ISR中做浮點運算是不明智的。

4) 與第三點一脈相承，printf()經常有重入和性能上的問題。如果你丟掉了第三和第四點，我不會太為難你的。不用說，如果你能得到後兩點，那麼你的被僱用前景越來越光明了。

```
/* 1.c */
int main()
{
int foo[] = {1};
int bar = 1;
return 0;
}
```

使用 gcc 將其彙編並以 intel 格式輸出組合語言檔案：

```
gcc -S -masm=intel 1.c
```

關鍵部分為：

```
mov     DWORD PTR [esp 8], 1
mov     DWORD PTR [esp 12], 1
```

esp 8 位置就是那個 int foo[]，esp 12 位置就是那個 int bar。可見，給 int 陣列的賦**值**時就像給一個 int 變數賦**值**一樣，並沒用指標來進行間接訪問，這個 int 陣列物件 foo 的記憶體地址在編譯時就確定了，是 esp 8；正如那個 int 物件 bar 一樣，它的記憶體地址在編譯時也確定了，是esp 12。

```
/* 2.c */
#include <stdlib.h>
int main()
{
int *foo = (int *)malloc(sizeof (int));
*foo = 1;
return 0;
}
```

彙編的關鍵部分為：

```
mov     DWORD PTR [esp], 4
call    _malloc
mov     DWORD PTR [esp 28], eax
mov     eax, DWORD PTR [esp 28]
mov     DWORD PTR [eax], 1
```

前兩句為 foo 分配記憶體空間，第三句將分配的記憶體空間地址**值**賦給 foo，foo 的地址為 esp 28，編譯時已知。下面是賦**值**部分，首先從 foo 那裡得到地址**值**，然後向這個地址賦**值**，這裡可以看出和給陣列賦**值**的差別，給陣列賦**值**時是將**值**直接賦到了陣列中，而不用從哪裡得到陣列的地址。
由上面可以看出，陣列更像一個普通的變數，編譯時就知道了其地址，可以直接賦**值**。