Ch1,2 Introduction、System Structures

（這一章好亂，隨便看看……）

一：何謂 OS：

總的來說，作業系統由一些軟體何任體的程式模組所組成，他：

1. 管理協調各應用程式間對資源的使用。
2. 方便對系統資源的使用（提供環境）。
3. 提高系統效率。
4. 保護記憶體中程式，不被其他程式破壞。

而站在使用者觀點，作業系統是種延伸機器(Extended Machine)

1. 使用者可以不用直接控制硬體。
2. 提供程式介面(porocedural interface)，讓使用者可以直接用 system call 來呼叫系統進行服務。
3. 交談式介面(user interactive interface)。
4. 多使用者時，作業系統必須提高資源使用率。
5. 對資源，系統協調專用(dedicated)形式或是非專用。

而站在系統觀點，作業系統是資源管理者(Resourse Manager)

1. 分配資源並且進行必要的管制。
2. 管理的資源，分為實體資源（印表機等）和虛擬資源（記憶體、信號機）。
3. 資源以多工的形式供使用者輪流使用，可分為間多工時空間多工。
4. 提供保護功能

Note：Bare Machine 是指硬體，Extended Machine 是指系統軟體、應用軟體

二：OS 的別名：

Resource allocator	Control program	kernel

三：OS 的組成元件：

打	P	Process Management	程序的建立、刪除、暫停、恢復、同步(critical section)、通訊(message ssing、shared memory)、死結處理
不	M	Memory Management	記憶體的監督、載入、分配、回收
死	S	Secondary Storage Management	可用空間管理、儲存體配置、磁碟排程
	I/O	I/O system Management	buffer、driver
讓他	F	File Management	檔案、目錄的建立、刪除、管理檔案的 mapping、backup
打	P	Protection	CPU Protection Memory Protection I/O Protection File Protection Other resources Protection
穩	N	Network	
死	C	Command Interpreter System	

四：CPU 的工作型態：

I/O Bound Jobs	CPU 等 I/O
CPU Bound Jobs	I/O 等 CPU

五：OS 的演進、類型：

Batch System：批次系統，常使用 Card-Reader 的方式來輸入資料，電腦同一時間只能執行一次工作。
Multi-Programming：mem 中有多個程序，當正在執行的程序正在 I/O，CPU 才執行另一個程序
Time-sharing 由 Multi-Programming 衍生而來，條件上多加了每個程序只能佔用一定的執行時間
Real-time：即時系統，若是要求很高的即時性系統，通常沒有 HD，以 RAM 做 HD，所以也沒有 Virtual Memory
Soft-Real-Time：軟性即時系統，用 Priority 來決定優先權，通常可以跟其他系統整合，例如跟 Time-Sharing 整合等。
Multi-Processing = Multi-Processor：多 CPU 又分為 symmetric(沒有主從關係)、asymmetric(主從關係) 有三優點： 1. 增加產量 2. 經濟考量 3. 提升可靠度
Clustering systems：以重複性提高可用性。 Asymmetric clustering：用一台機器熱待機，只監督工作，其他壞了馬上補上。 Symmetric clustering：所有機器互相監督。
Distributed Systems:：依賴網路，獨立且可能是異質的電腦組成（NFS、FTP、network、NOS、DOS 等名詞注意一下） 目的為： 1. Resourse sharing 2. Computing speedup 3. Reliability 4. Communication
即時嵌入式系統

Multi-Tasking	一個 user 可以同時處理多個工作 windows、Linux、Unix 都是
Multi-Thread Thread 又稱為 Light weight process	一個程序同時可以有多個 thread 在執行 windows、Linux、JVM 支援，傳統的 Unix 沒有，IBM 的 AIX，HP 的 HP-ux，Sun 的 Solaris 支援
Foreground(前景處理)	CPU 即時處理
Background(背景處理)	CPU 有空時才處理

六、User mode vs. Kernel mode

特權指令（Priviledged　Instruction）：常見的情況有：

1. 使用者模式切換到核心模式
2. I/O 控制
3. 設定計時器
4. 停止 CPU 執行
5. 變更中斷向量、抑制中斷
6. 更改 bound register

七、中斷(Interruptions)

1. 中斷的分類：

Soft Interrupt(也可以說是 Trap)	System Call	Soft
Internal interrupt 又稱為 Trap	Overflow Divide by zero	Hard
External interrupt	Machine Check Interrupt I/O interrupt Device(H/W Error)Interrupt	Hard

所以說 Trap 可以稱為由軟體產生的中斷

2. 中斷向量(interrupt vector)：中斷種類是有限的，所以可以用一個中斷向量表來儲存中斷常式的指標。
3. 中斷抑制與控制：可以用一個中斷遮蔽位元(Mask bit)來控制不要接受中斷，或是用 Priority 來控制。

八、I/O 控制→Polling vs Interrupt

1. **Polling：**又稱 Programmed I/O(PIO)或狀態核對(Status Checking)，其利用程式主動控制，OS 定時主動向各週邊設備輪詢(polling)。週邊設備如有需求就把 CPU 使用權交給該週邊設備，若無需求，則跳到下個週邊設備輪詢。適合低速資料的 I/O。缺點：1.CPU 浪費很多時間在測試 I/O 設備的狀態。2.資料傳輸經由 CPU，而不是直接由主記憶體與 I/O 進行傳輸

2. Interrupt I/O

1	2	3	4	5
CPU 收到 Interrupt 訊號	中斷目前工作，進行 context switch	透過 Interrupt Vector 尋找對應的 ISR(Interrupt Service Routine)，並執行之	完成 ISR	回到之前執行程序的狀態

九、I/O 控制→DMA

1. DMA 利用 DMA controller 代替 CPU 監督資料輸過程，此時 CPU 可執行其它工作。
2. Cycle Stealing：DMA controller 有資料傳送時，disable CPU 的 bus access，然後進行 memory access，資料傳送完後再 enable CPU 的 bus access。中斷的時間以一個 block 的傳輸為單位。

DMA 在 I/O 時，CPU 不需執行 ISR 動作

十、Buffering vs Spolling

Buffering	主要為了調節兩硬體不同的速度，以提升效能
Spooling	主要為了讓專用裝置(Dedicated Device)共用

Spool：Simultaneous Peripheral Operation On-Line（見下圖）

十一、常見結構：

1. **單調結構(Monolithic)：**由單一程式所構成的完整實體，如 Unix、MS-DOS。
2. **分層結構**
3. **微核心架構：**將核心模組化，不是必要的就用獨立的行程來實作，以減少核心大小，只留下基本的行程、記憶體管理和通訊等一些功能。而使用者和微核心間經由 Messege pass 來傳遞。
4. **模組化核心：**物件導向的程式設計，由一群中心模組組成，每個模組都是獨立的。系統視需要載入模組。
5. **虛擬機器(Virtual Machine)**：單一電腦硬體轉換成多個執行環境

   優點：系統間間完全隔離，保護容易且開發新系統容易。
6. **System Generation：**良身訂製的系統，針對不同硬體作不同組合，程式會測試硬體的組態，自動產生可以使用的 OS。

十二、開機時硬體動作

1. 靴帶式載入(Bootstrap program)：存於 ROM 或是 Firmware 的一小段程式碼，找出 kernel 並載入。
2. 兩步驟載入：由固定位置啟動磁區，然後用 bootstrap loder 載入作業系統。

十三、使用者作業系統介面

可分為**交談式介面**和**程式介面**，交談式介面讓使用者可以與系統直接溝通，程式介面提供行程呼叫以提供必要的服務。常見的交談式界面有：

1. 命令列介面或直譯程式：可選擇 Shell，
2. 圖形使用者介面
3. (恐龍本沒寫)：選單介面

（我討厭 Threads……）

一、Process

程式(program)本身是被動的，執行以後才是 process，

行程的產生可以用 fork()函數來呼叫，呼叫的行程叫做 Father process，而被呼叫的就叫做 Children process(詳細作法的在第六章)

1. Process 包含：
   1. **text section**：程式碼
   2. **program counter 與 register set**
   3. **stack**：**存放暫時性資料**
   4. **data section**：含有 global 變數
   5. **heap**：以便在 runtime 時可以動態配置記憶體
2. 行程的狀態可用下圖表示（有七個轉換時期）：

二、 PCB(Process control block)

每一個行程都對應一個 PCB，紀錄的資訊包括

1. **行程狀態(process state)**
2. **Program Counter**
3. **CPU 暫存器**：包含了 stack pointer、一般用途暫存器、狀況代碼或是程式狀態字組 PSW 等。
4. **排程相關資訊**
5. **記憶體管理資訊**
6. **會計資訊**：帳號、時間限制、用掉的處理時間等。
7. **輸出入狀態**

三、 內文切換(context switch)

1. 行程的狀態紀錄在 PCB 中
2. 在 context switch 發生時，如何傳遞參數(parameters)？
   →利用暫存器：量少但快
   →利用 Block 或是 Table：可以傳大量資料但 Overhead
   →用 Stack：傳比較大量資料，但進入 Kernel 時麻煩要切換

四、 行程間通訊

1、 **Shared Memory**

2、 **Messege Passing**：可分為 blocking 和 nonblocking，也稱為同步(synchronous)或非同步(asynchronous)

五、 Thread(又稱為 LWP　Light Weight Process)

1、 一個 Process 可以包含多個 Threads，他們共用

code section

data section

recourse(file、children process 等)

2、 Thread 有獨立的

Thread ID

Program Counter

Resgister set

Execution State

Run-Time Stack

3、 屬於同一個 Process 的 Thread 是合作關係，所以不用作 Protection。

4、 Thread 有一些優點

Responsiveness

Resource sharing

Economy

Utilization of Multiprocessor architectures (使用多處理器架構)

六、 Thread 的分類

1. **User Level Thread**
2. **Kernel Level Thread**

七、Thread Model 分類

1. **Many-to-One**：同一時間只能有一個 Thread 在系統中執行，因此不能讓多個 process 在 mutiprocesser 系統上平行執行。只要有一個 user Thread 被 block 住，整個 process 就會被 block。可用 user level thread libraries 在 user level 之進行切換，user level 平行性差。（ex. Solaris 的 Green threads library）
2. **One-to-One**：提供比 Many-to-One 更多平行性，一個 Thread 被 block 時其他 thread 還可以執行，可以在 mutiprocesser 系統上平行執行，User level 平行性佳，但佔用較多的 Resource。 (ex.WinNT、Win2000、OS/2)
3. **Many-to-Many**：是個彈性用法，相較於 One-to-One，

Many-to-Many 不用擔心產生太多 Thread，當一個執行緒暫停執行系統呼叫的時候，核心可以安排另一個 Thread 執行。（ex. Solaris 2、IRIX、HP-UX、True64 UNIX）

4. 還有一種兩層模式：屬於 Many-to-Many 的變形

LWP 包括：正在執行的 user thread 的 register set、memory 空間、account information 等（沒有 stack）。<<~什麼啦

八、Thread Library

1. **POSIX Pthread**
2. **Win32**
3. **JAVA Pthread**

Solaris2 的 Pthread (小強說全部記…隨便講的吧…)

1. 在 user 和 kernel thread 之間定義了一個 LWP,
2. 每個 process 都至少一個 LWP，一個或多個 thread 可以對應到一個 LWP，但同一時間一個 LWP 只能對應一個 thread。
3. 屬於 **Many-to-Many**
4. User Thread 分為 **bound** 和 **unbound** 兩種，bound thread 對應到固定的 LWP，適合 Real time。Unbound 則是從 LWP pool 中，找 LWP 來對應。
5. User level 包含了有 Thread ID、Program Counter、Resgister set、Stack、priority。
6. LWP 包含正在執行 user thread 的 rrgister set，memory account information，都是 kernel structure 且放在 kernel space
7. Kernel 包含　kernel register 副本、指向 LWP 指標、priority 與 scheduling information

九、Thread Cancellation

被取消的 Thread 叫做 target Thread，分為兩種不同的情況 asynchronous cancellation(非同步，立即終止)　和　deferred cancellation（延遲，週期檢查自己是不是被取消了）

十、Thread Pools

因為不能讓 Thtread 無限制產生，而且一直要有的動作一直產生花時間，所以用 Thread pool。例如伺服器中一個 Thread 被建立就被放到 Pool 裡面，當伺服器接收到要求的時候就從 pool 裡面抓，做完在放進去，這樣有兩優點：

使用現存的比產生一個新的 Thread 快

限制了任何時候 Thrrad 的個數

十、排班的觀念

1. 在 mutiprogramming system 中，有 n 個 process p_0 p_1 p_2 p_3…p_{n-1}，他們 I/O 的機率是 r_0 r_1 r_2……..r_{n-1}，則最大使用率為：$u = 1 - r_0 r_1 r_2 \ldots r_{n-1}$ <~ 代表扣掉全部都在做Ｉ/O
2. Scheduling 的基本單位：Process
3. CPU-I/O Brust cycle，CPU 動作和 I/O 動作交替出現，使用 CPU 的時間叫做 CPU brust，反之叫做 I/O brust

第一章提到過的兩種 Job 分類

I/O Bound Jobs	CPU 等 I/O
CPU Bound Jobs	I/O 等 CPU

CPU-I/O Brust cycle，CPU 動作和 I/O 動作交替出現，使用 CPU 的時間叫做 CPU brus

4. Scheduling Queues：在排隊的用一個 queue 串聯起來，很多地方都有這東西，Ready Queue、Device Queue、信號機也有。
5. **Long-term Scheduling：**長期排班，決定哪些狀態進入到 Ready Queue
6. **Short-term Scheduling:**短期排班，決定哪些狀態從 Ready 去 Run。
7. **Medium-term Scheduling：**對於某些不適合繼續執行的程式，把他們移出(swap out)記憶體，或是恢復被暫停的程式，將他們移入(swap in)，以平衡 I/O bound 和 CPU bound 或是系統發現某 process 影響效率之情況。
8. **Dispatcher(分配器):**將 CPU 的控制權交給由短期排程所

選定的程序，要進行的工作有：

1. 執行本文交換
2. 將執行模式變成使用者模式
3. 跳至適當的程式，開始執行

note：dispatch latency：分配程式終止一個程序並開始執行另一個程序所花費的時間

十一、裁決模式

1. 在以下狀況會有排班產生

其中如果沒有(2)，那這就是一個 multi-programming，有就是是個 Time Sharing

2. Non-preemptive vs Preemptive

	優點	缺點
Non-preemptive	適用 Soft real time 系統	文本切換次數多且要保留記憶體空間給程式，時間以及空間的 Overhead，**Starvation**
Preemptive	整體而言比較公平，反應時間好預測	Short job 等 long job 不適合 multi-user

3. **Arbitration**：仲裁法則，當有相同優先等級的時候，要怎麼決定是誰先進去？Random、Chronological（照時間）、Cyclic。

十二：CPU scheduling 的 criteria：

T	**Turnaround time**	1. 從程序提交到程序完成過程所花的時間 2. 程序等待進入記憶體，等待進入預備佇列、CPU 執行、以及 I/O 等運作過程的時間
W	**Waiting time**	在預備佇列中等待的時間
C	**CPU utilization**	CPU 的使用率
R	**Response time**	從程序提交到得到第一個回應的時間間隔
T	**Throughput**(生產量)	每單位時間完成的 process 量

計算上有幾個代號：C-Complele Time 完成時間

R-Response Time

A-Arrival Time

S-Service Time

R = C – A = R + S

W = C – A – S

十三、排班演算法：

FCFS 或 FIFO	▲先來先做，效率最差，平均 turnaround time、waiting time 都比較長，會造成 convoy effect(護航效應) △convoy effect：其它 process 都在等一個要執行很久的 process ↑都你做就好了啊。
SJF (shortest job first)	▲可插隊排程，不中斷目前正在執行的程序 △CPU 每次執行都挑選工作時間最短的 時間一樣長的，就用 FCFS 決定 △困難是不容易預知 process 的執行時間
SRT、SRJF (shortest remaining time first)	▲可插隊排程，會中斷目前正在執行的程序 △一有新的程序到達，就檢查新程序是否剩餘時間最短，是就先執行
HRRN(Highest-Response-Ratio-Next) 我想問你哪位啊？	Non-Preemptive，有 aging 的效果
Priority	▲根據 process 的優先權值決定執行順序，因此 SJF 可視為 Priority 排班法的特例，只是 SJF 是以執行時間或剩餘時間做為 priority △優先權一樣高的，就用 FCFS 決定 ▲priority 排程會有 starvation 的問題，可用 aging technique(老化技術)逐步提升久等的 process 的 priority
RR (Round Robin)	運用 time quantum(時間配額)或 time slice(時間分割)的方式，限定每個 process 的執行時間。 turnaround time 比 SRTF 長，但 response time 比 SRTF 好
MLQ (Multi Level Queue)	將 queue 分為多個等級，每個 queue 各自有各自的排班演算法，process 不能在 queue 間移動，因此會有 starvation 的問題
MLFQ (Multi Level Feedback Queue)	同上，只是應用 aging technology，把用太久的 process 降低 priority，等太久的提高 priority。 time quantum 越來越大，最後一層可以設為無限大，就是 FCFS 的效果。

※ 快去看一下排班相關的題目

關係圖：

十四、即時排班

1. 硬性即時排班：不可能用在分時系統
2. 軟性即時排班：Priority 的概念，重要的先做，但可能造成優先權反轉（系統資源被 Priority 低的佔住）

十五、Thread 排班

單的沒什麼好說，Multi-processor 可以分為

1. **Load Sharing** ：空的時候抓一個出來執行
2. **Geng scheduling**：one-to-one
3. **Dedicated processor assignment**：每個 Processor 出來的時候就看他有多少 Thread 來分配
4. **Dynamic scheduling**：動態

十五、效能評估

1. **Deterministic Modeling** (分析式評估-定量模式)：每一組數據算
2. **Queuing Models**(分析式評估-排隊模式)：以機率來算
3. **Simulation**(模擬)：設計一個模擬程式來跑
4. **Implementation**(實作)

Ch6 Synchronization

一、行程相關運作

1. fork()：建立一個 child process，child 的會回傳 0，parent 的會回傳 child 的編號。
2. exec(name,argc,envp) ：載入新的程式來執行(用在 Child)
3. wait()：等 Child 執行完繼續做自己的。
4. exit(status)：結束離開，回傳 status 給 parent。

二、一個程式碼範例 (小強說一定要會，那搞懂一下好了)

1. **parent 和 child 平行執行**

```
int main()
{
    pid_t pid;
    if(pid < 0)
    {   fprint(stderr,"Error");
        exit(-1);
    }
    else if(pid==0)
    {   exec("/bin/1s","1s',NULL);   }
    else
    { //parent 和 child 同時在執行
        exit(0);
    }
 }
```

2. **parent 等 childchild 做完自己繼續做**

```
int main()
{
    pid_t pid;
    if(pid < 0)
    {   fprint(stderr,"Error");
        exit(-1);
    }
    else if(pid==0)
    {   exec("/bin/1s","1s',NULL);   }
    else
    { wait(NULL);
      //這裡要到 child 做完了，parent 才可以繼續做
        exit(0);
    }
 }
```

三、行程間通訊

1. **Shared memory**：
   適合大量資料傳送
   適合緊密系統，不適合分散式系統
   那一定要有好管理機制
2. **Message Passing**：
   容易實作
   不會有資料上衝突
   適合傳少訊息
   適合分散式系統

有下列兩個功能：send、receive，
有直接傳或是透過 mail box 的方式

3. **blocking**（要等對方收到/送過來才能繼續執行）
4. **non-blocking**（接收或是只得到 NULL/送了就算）

四、平行結構、內隱平行結構

有四種改進方法：

1. Bernstein' Concurrency Condition：觀察出一些情況就可以平行做。
2. Loop Distribution：陣列中相同的變數，可以拿出來一起做
3. Tree Height Reduction：數學上的結合率等，降低運算樹的高度
4. Never Wait：有 processor 空的就做一下

五、Critical Section

Mutual exclusion	如果 process Pi 正在 critical section 中執行時，其它 process 均不允許在它們的 critical section 中執行
Progress	如果沒有任何程序在臨界區間，且有某個程序想進入自己的臨界區間，它必須可以進去，且決定過程不能被無限期的延遲
Bounded waiting	一個程序提出進入臨界區的請求後，等待允許的時間必需是有限的。Ex:若有 n 個 process 要進入臨界區，則最多等到(n-1)次之後必可進入臨界區

六、二個 Process 的軟體演算法

1. Dekker's 演算法（好像比較少考…）

處理元 i：

repeat

```
flag[i] = true;                          //舉旗了
  while ( flag[j] )                      //檢查對方
  {
      if ( turn != i )
      {
          flag[i] = false;               //你先啦
          while ( turn != i ) {skip; }   //還在對方就一直跑
          flag[i] = true;                //該我了
      }
  }
```

…Critical Section…

```
flag[i] = false;
turn = j
```

…Remainder Section…

until false;

處理元 j：

repeat

```
fag[j] true;                             //舉旗了
  while ( flag[i] )                      //檢查對方
  {
      if ( turn != j )
      {
          flag[j] = false;               //你先啦
          while ( turn != j ) {skip; }   //還在對方就一直跑
          flag[j] = true;                //該我了
      }
  }
```

…Critical Section…

```
flag[j] = false;
turn = i
```

…Remainder Section…

until false;

2. Dekker's 演算法是否滿足

滿足 mutual exclusion	兩個 process 要同時進入 C.S.，Pi 需要 flag[j] = false 這時需要 pj 進入 while 中把自己設為 false => **turn = j** ，而 Pj 需要相反的，**turn = i**，turn 不會同時 = i 和 j，故成立。
滿足 progress	Pj 在其他段，Pi 可進入。 Pj 在 C.S 時 Pi 想進入，等 Pj 做完一次，這時 turn = i => 可以連跳兩個 while 出去(**中間 Pi 會卡在最上層 while，但這時他的 flag 是 true，造成 Pj 下一次得到 CPU 排班，會進入 while 把該改的改一改讓 Pi 進入**)，滿足。
滿足 bounded waiting	Time quantum 沒用完 Pj 可以進去很多次，但用完以後 Pi 會卡一次，接下來 Pj 就進不去，Pi 進去，不會 bounded waiting。

3. Peterson's 演算法 (要背喔)

處理元 i：

repeat

```
flag[i] = true;
turn = j;
while(flag[j] && turn ==j){skip;}
```

…Critical Section…

```
flag[i] = false
```

…Remainder Section…

until false;

處理元 i：

repeat

```
flag[j]= true;
turn = i;
while(flag[i]&& turn == i ) {skip;}
```

…Critical Section…

```
flag[j] = false
```

…Remainder Section…

until false;

4. Peterson's 演算法是否滿足

滿足 mutual exclusion	turn 要同時 = i 和 j 才能讓兩者同時進入，不可能，故成立。
滿足 progress	Pj 在其他段，flag[j] = false，Pi 可進入 Pj 在 C.S 時 Pi 想進入，下一次進來的時候，turn = i，這時就卡住了因為 flag[j] && turn ==j，而 pi 就可以進去。
滿足 bounded waiting	Pi 只要等 Pj 下一次進來，把 turn 設給 i，就可以進去了。不會 bounded waiting。

七：N Processes 的解決方案：

1. Eisenberg and Mc Guire's algorithm:

```
    var turn : 0..n-1;
    flag : array [0...n-1] of (idle, want-in, in-cs);
    initialize all flag[i] as idle,
    initial value of turn is immaterial.
    { for pi of n processes }
 1  repeat
 2   repeat
 3    flag[i] := want-in;
 4    j := turn;
 5    while j <> i do
 6      if flag[j] <> idle then j := turn
 7      else j := j + 1 mod n;
 8    flag[i] := in-CS;
 9    j:=0;
10    while ( (j < n) and ((j=i) or (flag[j] <> in-CS)) ) do
11      j := j + 1;
12   until ( (j >= n) and ((turn = i) or (flag[turn] = idle)) );
13   turn := i;
14  CS
15   j := turn + 1 mod n;
16   while ( (j <> turn) and (flag[j] = idle) ) do
17   j := j + 1 mod n;
18   turn := j;
19   flag[i] := idle;
20  non-CS;
21 until false;
```

2. Eisenberg and Mc Guire's algorithm 是否滿足

滿足 mutual exclusion	情況一：某程序已在 CS 中，此時新進的程序會在 5~7 列轉圈圈 情況二：假設程序 1、5 同時執行，此時 turn=0，若兩個程序同時通過了 5~7 列的檢查，則在 10~11 列一定會有一個被攔下來
滿足 progress	沒有程序在 CS 裡時，任意程序都能進得了 CS
滿足 bounded waiting	假設 n=5，而提出申請 CS 的順序是 0→4→1→2→3，則觀察一下程序的特性，雖然程序 4 是第 2 個提出的，但當 0 在 CS 中，而 1,2,3 也陸續提出申請 CS 時，0 一離開 CS，1,2,3 因為程序編號比 4 小，所以很可能(但不一定)先獲得執行權，但由於列 17 的作法，因此程序 4 一定會在 n-1 次的等待時間內獲得執行權

3. Bakery algorithm 麵包店演算法

```
    var choosing: array [0..n-1] of boolean;
    number : array [0..n-1] of integer;
    initially: all entries of choosing are false
    number = 0
    notation :
    (i) (a,b) < (c,d) if a < c,or if a = c and b < d
    (ii) max (a0, a1, a2, ... , an-1) is k such that k >= ai ; for i=0,1,...,n-1
    { for pi of n processes }
 1  repeat
 2   choosing[i]:=true;
 3   number[i]:=max (number[0],number [1],...number[n-1])+1;
 4   choosing[i]:=false;
 5   for j=0 to n-1 do
 6    begin
 7      while choosing[j] do no-op;
 8      while number[j] <> 0 and (number[j],j) < (number[i],i) do no-op;
 9    end;
10  CS
11   number[i] := 0 ;
12  non-CS
13 until false;
```

4. Bakery algorithm 是否滿足

滿足 mutual exclusion	由列 7、8 負責；其中列 7 到底有什麼用途呢?詳述如下： 假設程序 3 正在執行 max(……)+1 的運算，且因故做得特別久，而此時程序 4 已完成選號進到列 6~9。假設程序 4 拿到了 10 號，且因為沒有列 7 的檢查，程序 4 通過了列 6~9(因為此時 number[3]還是 0)；程序 4 進到了 CS 此時程序 3 獲得一連串的執行權，且當時因為跟程序 4 同時在選號，意即程序 3 在執行 max(……)+1 的動作時，已讀到了 number[4]，而當時 number[4]還是 0 因為程序 4 也正在取號，所以程序 3 獲得一連串的執行權時，也取到了 10 號。於是程序 3 也會順利進到 CS，所以如果少了列 7 會違反 mutual exclusion
滿足 progress	會依照 number[i]的大小進入，OK。
滿足 bounded waiting	最多等到 n-1 次就可以進入，OK。

Bakery 演算法並不保證兩個 process 不會收到同樣的號碼，如果 Pi、Pj 取到同號，則若 i<j，Pi 會先取得服務。

八、硬體演算法

1. Hardware instruction Support Solutions

```
function Test-and-Set(var target:Boolean):Boolean;
begin
        Test-and-Set:=target;
        Target:=true;
end;

//注意。var target 是傳址呼叫
```

```
Pi 之程式：Lock 初值為 false
repeat
 while Test-and-Set(lock) do no-op;
        CS
 Lock:=false;
        Remainder section
until false;
```

此程式滿足 Mutual Exclusion，但並不滿足 Bounded Waiting

```
    Var waiting:array[0……n-1] of boolean;
    lock:Boolean;

    var j:0….n-1;
    key:Boolean;

    repeat
 1   waiting[i]:=true;
 2   key:=true;
 3   while waiting[i] and key
 4   do key:=Test-and-set(Lock);
 5   waiting[i]:=false;
 6  CS
 7   j:=i+1 mod n;
 8   While (j<>i) and (not waiting[j]) do j:=j+1 mod n;
 9   If j=i then lock=false
10   else waiting[j]:=false;
11  Remainder section
12 Until false;
```

列 3~4	若 Lock=false，表示沒有程序在用 則一執行列 4 之後，就會順利進入 CS，而此時 Lock 變成 true 而由於 Test-and-set 函數只會 set Lock=true，所以其它程序會在 Lock 等於 true 時，一直在列 4 繞圈圈
列 9	如果都沒有別的程序，而且下一個程序是自己
列 10	由於其它所有在等待的程序都在列 3、4 繞圈圈，所以不能去動 Lock 變數，因為 Lock 變數一動，可能會有一狗票程序衝進來，因此從 waiting[j]變數下手，因為每個程序各自擁有一個，只要讓 waiting[j]=false，它就會結束列 3 進到 CS；也因此目前在 CS 中的程序可以指定下一個要執行的程序，如此可以達成 Bounded waiting

2. Swap 指令

```
Procedure Swap(var a,b:Boolean)
Var temp:Boolean;
begin
    temp:=a;
    a:=b;
    b:=temp;
end;
```

```
Pi 之程式：lock:公用變數，key:局部變數，初值皆為 false
Repeat
   key:=true;
   repeat
      swap(lock,key);
   until key:=false;
CS
   Lock:=false;
Remainder section
until false;
```

	第一個執行的程序，lock=false，key=true，所以會通過 swap 進到 CS，而當第一個程序一進入 swap，公用變數 lock 就變成 true，所以其它程序都會在 swap 那列繞圈圈，一直到 CS 完成把 lock 改成 false，才會讓下一個 process 進入 CS，但這個做法並不保證 Bounded Waiting

3. Disable Interrupt 指令：程式碼就不寫了，精神就是進入 C.S 的時候就不會被中斷，不過這個方法會造成問題，C.S 沒寫好出不來就完蛋了，或是 multi-processor 也不能且這必須是個特權指令。

九、Semaphores 信號機

1. 有些 semaphore 不需使用 busy waiting 的方法來同步

2. semaphore 是一個整數變數，提供兩種 atomic 運作：**wait** 與 **signal**，或稱 **P(wait)**與 **V(signal)**。wait、signal 的定義如下：

```
標準 semaphore
wait(S)：while s<=0 do no-op;
                S:=S-1;
Signal(S)：S:=S+1
```

```
Pi 程式：
semaphore:mutex=1;
repeat
  wait(mutex);
        Critical section
  signal(mutex);
        Remainder section
until false;
```

3. semaphore 可用來解決各種同步問題。例如 P1 程序中有 S1 敘述，P2 程式中有 S2 敘述。假設要求 S2 只能在 S1 執行完後再執行。要完成這要求，我們讓 P1 與 P2 共享一個 synch 號誌(synch 初值為 0)，做法如下：

```
P1：
        S1;
        Signal(synch);
P2：
        Wait(synch);
        S2;
```

這樣即可確保 P2 在 P1 執行過 S1 後才執行 S2；

4. Spin Lock(Busy-Waiting)、Suspend Lock：

1. Suspend、wakeup 的方式：為了克服 busy waiting 的方式，將要等待進入 CS 的 process 予以停滯(block)，停滯可將一個處理程序送入號誌對應的等待佇列之中，且將該 process 的 state 改為 waiting state。之後系統的控制便交由 CPU 的排程程式，由它來選擇另一個處理程序運作。
2. 實做：

```
Type semaphore = record
        Value：integer;
        L：list of process;
End;
Var S:semaphore;
每個號誌均有一個整數值及一佇列。當一個處理程序必須等待一個號誌時，它就加入處理程序的佇列。Signal 運作則可以移走等待佇列中的一個處理程序，並喚醒該處理程序。
運作方式如下：
```

Counting semaphore

```
wait(S):
   S.value = S.value-1
   if S.value < 0 then
   begin
       add this process to S.L;
       block(p);
   end

signal(S):
   S.value:=S.value+1;
   if S.value<=0 then
       begin
       remove a process P from S.L;(為首的)
       wakeup(P);
   end;
Value 的絕對值就是在等待號誌的 process 數目
這個做法是用串列把等著要進入 CS 的 process 串起來，再一個一個叫醒
```

5. 二元信號機：這個信號機只能是 0 或 1，運算如下

```
wait(S)
   if S.value = 0 then;
       add this process to S.L;
   S.value = 0;
Signal(S)
   S.value = 1;
       remove a process P from S.L;（為首的）
```

好像會考用二元模擬普通的：

```
Init S1 = 1,S2 = 0,S3 = 1,C counting semaphore  初值
genwait (C)
   wait(S3);
   wait(S1);
   C = C -1;
   if C < 0 then begin
       signal(S1);
       swait(S2);
       end;
   else slgnal(S1);
   signal(S3);
```

```
gensignal(C);
    wait(S1);
    C = C+ 1;
    if C < 0 then signal (S2);
    signal(S1);
```

十、Moniter 監督器

1. 是一種較高等的結構，含有較多功能
2. 在 Moniter 之中可以宣告變數，但是 process 要呼叫必須經過 entry procecdure 才可以呼叫，提供 data abstract 的估能
3. 要維持 mutual exclusion，所以通常用一個 FIFO 的 queue 來實做
4. 系統用 shared memory 的方式來通訊
5. 系統定義 condition varible，提供 wait 和 signal 兩種 primitives，每一個 varible 有一個 queue(FIFO 或是 priority)
6. 一個簡單的範例

管理一個專用裝置：

```
type resourse-allocation = monitor
var busy: Boolean;    x:condition;

procedure entry acquire(time : integer);
   begin
      if busy then x.wait(time);
      busy = true;
   end'
procedure entry release;
   begin
      busy = false;
      x.signal;
   end;
init
begin
      busy = false
end
```

7. Moniter signal 別人以後，自己怎麼辦？這裡有三種選擇(1)signal and wait(2)signal and continue(3)signal and exit，
8. 如果說要用 semaphore 的話：

(1)每一個 entry process F　改寫如下（一次只能一個進去）

```
wait(mutex)
   ….
signal(mutex);
```

(2)每一個 x.wait 改寫如下

```
x_count = x.count +1 ;
if next+count > 0 then signal(next);
else signal(mutex);
wait(x_sem);
x_count = x_count – 1;
```

(3)每一個 x.signal 改寫如下

```
if x_count > 0 then
   begin
      next_count = next count + 1 ;
      signal(x_sem);
      wait(next);
      next_count = next_count - 1 ;
   end;
```

十一、重要平行問題

1. Bounded Buffer Producer-Consumer Problem

利用號誌來完成生產者-消費者問題，共用變數：

mutex：CS 互斥用，初值為 1

empty：判斷 buffer 是否為空，初值為 n

full：判斷 buffer 是否滿了，初值為 0

```
生產者：
repeat
…
produce an item in nextp
…
wait(empty);
wait(mutex);       //lock CS
…
buffer[rear] = new item;
rear = (rear+1)mod n;
…
signal(mutex);       //release CS
signal(full);
until false;
```

```
消費者：
repeat
wait(full);
wait(mutex);       //lock CS
…
remove an item from buffer to nextc
…
signal(mutex);       //release CS
signal(empty);
…
get a item from buffer[foint]
front = (front +1)mod n;
…
until false;
```

2. Reader/Writer Problem

為了避免同時有 2 個以上程序對相同物件進行寫入，必需保護共享物件，當有寫入者要更新共享物件時採取互斥。

Var mutex=1

wrt:semaphore=1

readcount:integer=0

```
寫入者：
wait(wrt);
…
writing is performed;
…
signal(wrt)
```

```
讀取者：
wait(mutex);
readcount:=readcount+1;
if readcount=1 then wait(wrt); //加之前就少於 1
signal(mutex);
…
reading is performed
```

```
…
wait(mutex);
readcount:=readcount-1;
if readcount=0 then signal(wrt);
signal(mutex)
```

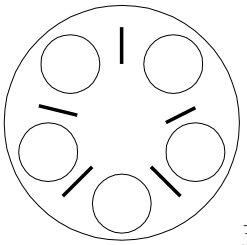
if readcount=1 then wait(wrt);

第一個讀取者要負責上鎖，避免寫入者做寫入動作

if readcount=0 then signal(wrt);

最後一個讀取者離開，要開鎖，才能讓寫入者寫入

3. 哲學家用餐問題



五個人、五根筷子

哲學家問題也是一種程序同步的問題，在上述的情況中如果每人都拿起一根筷子就死結了

```
Repeat
   wait(chopstick[i]);
   wait(chopstick[i+1 mod 5]);
      ……
      eat
      ……
   signal(chopstick[i]);
   signal(chopstick[i+1 mod 5]);
      ……
      think
      ……
until false
```

上述用 semaphore 解決了 CS 的問題，不過還是存在著 deadlock、starvation 的問題

▲deadlock 的避免方式：

1. 最多脫許 4 個人同時提出用餐的 request
2. 只有當身邊的兩根筷子都能用時，才允許拿筷子
3. 奇數哲學家拿左邊，再拿右邊，偶數先拿右再拿左

△但還是會有 starvation 的問題，該如何避免?

十二、並行語言結構 Critical Regions

1. 有兩種 **unconditional critical region、conditional critical region**

2. 用法：

region **a** do segment　代表無條件時

region **a** when **condition** do statement　代表有條件時

3. 一個用 Critical Region 解決 Bounded buffer 的例子：

```
producer：region buffer when count < n do
放東西進去;
count = count +1;
consumer region buffer when count > 0 do
拿東西出來;
count = count - 1 ;
```

4. 用 **semaphore** 來模擬：

```
region a when B do S;
初值設定：mutex = 1,sleep = retry = 0;
                SeleepCount = RetryCount = 0;
wait(mutex);
while ( not B )
{
   SleepCount ++;
   if ( RetryCount > 0 ){ signal(retry); }
      else {signal retry; }
   wait(sleep);
   SleepCount --;
   RetryCount ++;
   if ( SleepCount < 0 ){ signal (sleep);
      else{signal(retry);}
   wait (retry);
   RetryCount --;
}
……S……
if ( SleepCount > 0 ){signal (sleep);}
   else if ( RetryCount > 0 ){ signal(retry);
   else{signal(mutex);
```

註：只要有 process 執行完，就讓所有 Process 都到 Retry 檢查（第一個檢查對的就進去了）

十三、ADA 的會晤(Rendezous)

1. 是隨機選擇的，如果有兩個條件都滿足，那會隨機選一個來做，不像一般語言結構一定是上面的先。

```
begin
   loop
      select
         when count < n =>
         acceptinsert( it : item )
         ……存入 buffer……
         count = count +1 ;
         or when count > 0 = >
         acceptremove(it : out item )
         ……取出 buffer……
         count = count – 1;
      end select
   end
end
```

十四、不可分割的交易

1. 系統模式：執行單獨一項邏輯功能的一群指令，叫做一個 **transaction**（交易），讓整個交易都要完成或是都不進去做，叫做 **transaction atomically**
2. **Log Based Recovery**：把 transaction 中的資訊寫入 stable storage 中，開始時，把**<Ti start>**寫入 log，commit 時把**<Ti commit>**寫入 log。
   這會讓每次動作有 overhead，但還是值得
   用 **undo(Ti)**和 **redo(Ti)**兩種動作來做，如果有 Start 沒有 commit 就做 undo，有 commit 就做 redo
3. **Check Point**：上面的 undo 及 redo，大部分的 redo 動作其實整個 transaction 都已經做完了，這樣 recovery 很浪費時間。用一個 checkpoint 把做好的都輸出到 stable storage 中，之後從後面檢查就好了，以節省 recovery 的間。
4. **Serializability**（可串列化）：有些 transaction 之間並沒有 mutual exclusion 的問題，可以平行執行來提高效率。

5. **Lock Protocal**：分兩種 **Sahared Lock** 和 **Exclusive Lock**
   還有一個雙向上鎖節定（**Two phase lock protocal**），要求每一筆交易一兩種 phase 發出上鎖和解所要求。
   分為 **growing phase** 和 **shrink lock**。
6. **Timestamp**：每一個 transaction 執行前先給一個 **TS(Ti)**，越來越大。每一筆資料 Q 記錄 **W-timestamp**、**R-timestamp**，如果

   Read 的控制：

   TS(Ti) < W-timestamp(Q)：不能 read 要 rollback

   TS(Ti) >= W-timestam(Q)p 可以 read

   Write 的控制：

   TS(Ti) < R-timestamp(Q)：不能 write(Q)

   TS(Ti) < W-timestamp(Q)：也不能 write(Q)

   TS(Ti) >= W-timestamp(Q)

   且 TS(Ti) => R-timestamp(Q)：可以 write(Q)

Ch7 DeadLock

一、DeadLock 的定義：

一組被停滯的程序中，因互相等待而形成的現象

二、DeadLock 的必要條件：

Mutual exclusion	同一時間一個資源只能被一個 process 佔用
Hold and wait	Process 已持有資源，並在等待被其它 process 佔用的資源
No preemption	資源要由 process 自願釋放，不能被其它 process 強取
Circular wait	持有與等待的 processes 形成一個循環

三、四種解決方法

DeadLock Prevent 死結預防

DeadLock Avoidance 死結避免

DeadLock Detection 死結偵測

DadLock Recovery 死結回復

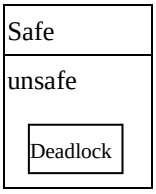
四、DeadLock Prevent：

只要使死結的四個必要條件不完全成立，即可預防死結

Mutual exclusion	非共享資源的互斥條件無法打破，如印表機，天生就不可共用
Hold and wait	1. 一開始就全配：Process 要取得所需的全部資源才能開始執行，否則要空手等待 (此方法是一種預先資源分配的方式) 2. 吃完了再夾菜：process 要在沒有佔用資源的情形下才能請求資源 缺點： 1. 會發生 starvation，因為某些資源總是被別人取得 2. 資源利用率低，因為 process 會用到該資源就要先分配，但 process 可能要過一段時間才會用到它
No preemption	也就是允許插隊的意思，做法如下： 請求資源時： 1. 沒人在用→給 2. 有人佔用，但目前是等待狀態→搶 3. 不能給→我等，而別人此時可以來搶我 適用時機： 狀態容易保存、被搶奪後容易恢復的資源，如 register、memory，但不適用於 printer、磁帶機
Circular wait	1. 將資源遞增編號 2. 規定：process Pi，resource Ri ▲如果 Pi 目前未佔用資源→可以請求任何編號的資源 ▲如果 Pi 目前佔用資源 Ri→則它只能申請比 Ri 編號大的資源 ▲如果 Pi 目前佔用資源 Ri→而申請比 Ri 編號小的 Rj 資源，則要先釋放目前佔用的資源編號大於 Rj 的所有資源 3. 證明： 反證法：假設使用上述原則，還是會發生 circular waiting，那麼必存在著 $R_i \rightarrow P_i \rightarrow R_{i+k}$ 其中 i,k>0 $R_{i+k} \rightarrow P_j \rightarrow R_i$ 其中 j>i (note：上述的表示法是 Pi 佔用 Ri 資源並且等待 R_{i+k} 資源) 但前面已規定 i 必需小於 i+k 狀況一才能存在，而 i+k 必需小於 i 狀況二才能存在，因為不存在這樣的 i,k 值，所以此法能避免 circular wait

五：DeadLock Avoidance：

1. 何謂安全狀態



Safe state 是指可以找到一組執行順序完成執行所有的程序（**safe sequence**）則是找不到這樣的執行順序，但不一定會發生死結，所以死結只是 unsafe state 的一部分

為什麼 unsafe 不一定 deadlock 呢?

2. **多副本：Banker's Algorithm**

先來個 **Safe algorithm**：

資料結構：5 個 process，3 種 resource

	Allocation 已分配資源			Max 最大需求資源			**Need**=Max-Allocation 尚需多少資源			Available 可用資源		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0												
P_1												
P_2												
P_3												
P_4												

時間複雜度：$m*(n(n+1)/2)=O(mn^2)$，其中 n 表 process，m 表 resource

證明：

步驟 1：Need=Max-Allocation，初始化一個 m*n 陣列→O(mn)

步驟 2：

檢查是否 need<=available→最多檢查 n 個

檢查是否 need<=available→最多檢查 n-1 個

…

檢查是否 need<=available→最多檢查 2 個

檢查是否 need<=available→檢查最後 1 個

1+2+3+……+(n-1)+n

Resourse-Requist Algorithm：先把要的東西扣掉，再繼續去找有沒有找到安全序列。

3. **單副本**：如果每個資源的 available 都是 1 時，可用圖形的方式(claim edge)（虛線），檢查圖形有沒有發生循環，有的話表示會發生循環

六、DeadLock Detection：

1. 多副本：基本上同 Banker，但把 **Need** 改成 **Reuist**，只看眼前。
2. 單一副本：把資源轉成圖，檢查有無迴圈，但可以有改進的方法：

3. 時機：

當有程序請求資源卻沒有立即得到允許時

固定時間間隔執行

CPU 的使用率低於 40%時

4. **這個公式背起來！**

七、DadLock Recovery：

1. 人工回復
2. 終止處理元資
3. 資源回收
4. Rollback

八、分散式系統的

1. 線性資源配置

2. Banker

3. Timestamp：分為 **Wait-Die** 和 **Wound-Wait**

Wait-Die：老的等年輕的：Pi 要 Pj 的資源，如果 Pi 小於 Pj，Pi 等，否則 Pj 倒退並且讓出資源。（越老越容易要等）

Wound-Wait：年輕等老的：如果 Pi 要 Pj 的資源，Pi 小於 Pj，則 Pj 倒退（會被 Pi 搶走），否則 Pi 等候

4. 死結偵測：

集中式偵測

階層式偵測

分散式偵測

九、消耗性資源的死結偵測

Ch8,9 Virtual Memory

一、基本概念

1. Address Binding：

Compile time	如果此時知道程式要擺在那裡，就產生 absolute code，否則就產生 relocatable code 例如.com 檔是 absolute（絕對）的，而.exe 可以是 relocatable
Load time	程式編譯要產生重定碼，在載入的時間才會有位址
Execution time	如果執行的時候會被移動到記憶體另外的地方那就要再執行的時候才載入

2. Logical address（Virtual address）和 Physical address：通常前兩種 bind 的邏輯位址和實體位址是一樣的，第三種則不同。由 **MMU（Memory management unit）**管理

3. Dynamic Loading：當副程式被叫用到時，程式才載入到記憶體中；運作方式：一開始只有主程式被載入，當副程式被呼叫時，系統檢查程式是否已在記憶體中，若不在，則利用可重新定位 linker 來將此副程式載入，然後將控制權交給這個副程式。

4. Dynamic Linking：Dynamic loading 是執行時才「載入」，Dynamic Linking 是執行時才「連結」。
載入跟連結的差異是，載入是指該段程式還不在記憶體中，而連結則是已在記憶體中

1. 執行程式中會有一段程式碼(stub)，它用來表示如何找到記憶體中的函式庫(library routine)
2. 當執行到這段 stub 時，會把自己代換成那個函式庫所在的起始位置，並跳到那個位址開始執行
3. 優點是數個程式可以共用同一個函式庫

ex：MS 的.dll 檔，Linux、Unix 的.so 檔

5. Overlay：Dynamic loading 的一種技巧，程式執行時只保留當時所需的指令及資料，當需要用到其它指令時，就把這些指令載入到已經不再使用的指令所佔用的空間中。

6. Swapping：

swap in：將程式由記憶體移至備用儲存體上

swap out：將程式由備用儲存體移至記憶體中

▲有一種以優先權為基礎的 swapping，若有一個優先權較高的程序要執行，而記憶體空間不足，此時為了載入此優先權較高的程序，可以將優先權較低的程序先 swap out，當優先權高的程序執行完後，再將優先權低的程序 swap in 繼續執行。這種就是 swapping，又稱之為 roll in/roll out。

二、連續記憶體配置

1. 功用：將記憶體分割成數個部分，分別由不同的記憶體來使用，用 **Bounded Register** 和 **Base-Limit Register** 來保護。

2. 固定多重分割（Fixed）：固定大小的分割交給不同的 Process 使用。

3. 可變多重分割（Variable）：恐龍本主要講這個，有三種配置方法：

First-fit

Best-fit

Worst-fit

4. 多重分割問題：Fragment 分為 **External fragment** 和 **Internal fragment**，最常是用 **compaction**（聚集）來解決。

5. 計算記憶體空間使用率：

三、Page

1. 相關名詞：

Frames	實際記憶體被分成許多固定大小的區塊
Pages	邏輯記憶也被分成許多「同樣大小」的區塊
Shared Pages (共用頁面)	Reentrant：執行中的程式碼可跟別人共用的稱為可重入式程式碼，也叫做純碼(Pure code)

2. Paging 的優點：

1.解決 External fragmentation
2.頁面可以共用
3.可設定頁面為 Read only，頁面保護較容易

3. Paging 的缺點：

會有內部碎裂	為了減少內部碎裂，可將 page size 縮小，但會造成 page table 變大，反而佔據空間
Logical address to Physical address 的轉換時間	
需要 HW 的支援，右列是三種實做法式，以第三種為最佳	1.用一組暫存器來儲存，可以快速完成 Logical address to Physical address 的轉換時間，但限制是分頁表(亦即分頁數量)要很小
	2.把分頁表放在主記憶體中，使用 PTBR：**Page Table Base Register** 分頁表基址暫存器，**來指向分頁表在記憶體中的位址**。這樣內文轉換時只需改變這個暫存器，可縮短處理轉換的時間 做法：CPU 去 PTBR 中讀取 page 所在的記憶體位置，讀出 frame number 後，再+page offset→所要的位置 因此一共要讀取主記憶體 2 次
	3.運用 **TLB(Transaction lookaside buffers** 或稱 associative register)，根據電腦的 locality 特性，TLB 做為 Page Table 的 cache，可大幅提升記憶體存取的效率

4. 實做方式：

Page table 裡的資訊：

Valid bit

頁面載入的 frame 編號（位址）

在磁碟中的備分儲存位址

可能還會有一些 reference bit、dirty bit 等資訊

四、Segmentation

1. Segmentation 的優點：

1.沒有 internal fragmentation
2.區段可以共用
3.可設定區段為 Read only，區段保護較容易

2. Segmentation 的缺點：

<table>
<tr><td>會有外部碎裂</td><td></td></tr>
<tr><td colspan="2">Logical address to Physical address 的轉換時間</td></tr>
<tr><td rowspan="3">需要 HW 的支援，右列是三種實做法式，以第三種為最佳</td><td>1.用一組暫存器來儲存，可以快速完成 Logical address to Physical address 的轉換時間，限制是分段不能太多</td></tr>
<tr><td>2.類似 paging 的方法 2，要用到 STBR(Segment Table Base Register)、STLR(Segment Table Length Register)

因此一共要讀取主記憶體 2 次</td></tr>
<tr><td>3.跟 paging 的做法類似，運用 TLB</td></tr>
</table>

3. 實做方式：

Segment Table 有下列資訊：

Valid bit（Resident bit）

在輔助記憶體的儲存位址

分段的長度

分段在記憶體的起始位址

保護位元（使用的權利 rwx）

對應步驟（因為要檢查一些東西）：

檢查分段編號是否正確（s < STLR）

算 entry（ i = STBR + s ）

把 segment table 讀出來

檢查動作是否符合權力

檢查 d<ST[i].limit，防止存取超出分段

如果 r = 0 代表要配置空間並且從輔助記憶體載入

五：Paged Segment

六、Virtual Memory 的意義、優點

1. 作法：CPU 送出虛擬位址，透過 memory map 對應到實體記憶體的位址。

2. 意義：允許執行的 process 不必全部 load 到記憶體中，亦即實際執行中的 process 大小可以大於實際記憶體的大小。

3. 優點：

1.程式可以不受實體記憶的可用空間限制
2.每個 process 只佔有少許的記憶體空間，提升 CPU 的使用率以及輸出量
3.process 載入記憶體的 I/O 時間縮短(一開始不必全部載入)
4.程式撰寫更容易，可以不用 overlay 的技術

4. 擷取策略：Pure paging、**Demand Paging**、Pre-fetch

5. Page Fault

1. 當執行中的 process 要存取某個沒有載入到記憶體中的頁面時，就會發生 page fault 中斷。
2. Page Fault 的處理：

檢查 process 的內部表(通常在 PCB 中)，以確定記憶體存取是否合法
若 interrupt 是因為不合法的存取造成，則終止 process，否則表示是由 page fault 造成
從主記憶體中找尋一個 free frame(若沒空間則執行 page replacement)
將所需的頁面讀入新分配的 frame 裡
載入頁面後，修改 process 的內部表，以及修改分頁表，將 invalid bit 改為 valid bit

七：Page Replacement

1. 當記憶體內沒有可用的頁框時，尋找一個犧牲頁面，將頁框裡的內容寫到磁碟上空出這個頁框，然後更正分頁表
2. Page Replacement Algorithms：(常考計算題)

FIFO		X
Optimal	當需要從記憶體中找出一個 frame 來替換時，選擇未來最久才會被參考存取的那個 frame 此法保證不會有 **Belady's anomaly**，page fault 次數最少，但有實作上的困難，因為無法預知未來(跟 SJF 遇到的困難類似)	O
LRU(Least Recently Used)	置換最長時間沒被用到的 frame 實作方式： 1. 計數器 2. 堆疊	O
LFU(Least Frequently Used)	每個 frame 一個 counter，剛載入時設為 0，參考到一次就加 1，替換時選擇 counter 最小的頁面替換。類似 LRU，但是因為剛進入的 counter 很低，所以會一直被換掉。 改進：經過一段時間可以往右移一個 bit（或是-1，就是 aging）	X
Second Chance	以 FIFO 為基礎，選擇一個 FIFO 頁面後，檢查參考位元，如果為 0，就替換這個頁面，若為 1，則修改為 0，但這次不做替換。如果某個頁面經常被使用到，參考位元就會一直保持在 1。 改進：用一個 circular queue。	O
Enhanced Second Chance (Not used Recently)	(reference bit , modify bit) (0,0)沒用，沒改 (0,1)沒用，有改 (1,0)有用，沒改 (1,1)有用，有改 替換的優先順序為上列四項由上而下	O
History Register	選暫存器值小的來替換	O
MFU(Most Frequently Used)	每個 frame 一個 counter，剛載入時設為 0，參考到一次就加 1，替換時選擇 counter 最大的頁面替換	X
Random		X

3. Belady's anomaly：一般而言，page frames 越大，page fault 的機率就會降低，但對於某些頁面替換演算法，page fault 的發生次數會隨著 page frames 變大而增加，此現象稱之為 Belady's anomaly。

例：1,2,3,4,1,2,5,1,2,3,4,5，frames 從 3 個→4 個

七、 Frame 配置法則

1. **Global replacement**：在執行 page replacement 時，所有記憶體中的 frame 都可做置換，可以配合可用頁框池（Free frame pool）
2. **Local replacement**：局部置換，在執行 page replacement 時，只能從自己所擁有的 frame 中做置換，要配合以下原則：**最少頁框數、平均分配、比例分配、程式優先權**。因此在分配工作的時候要考慮是否有足夠的空間分配給新程式

八、Thrashing

1. 一個程序如果花在 page replacement 的時間比執行時間還多，就表示它正處於猛移現象之中
2. 形成的原因：

page fault、page replacement、multiprogramming 的 degree

1.在 multiprogramming 的環境下，採用 global page replacement algorithm(即不管替換的頁面屬於那個 process，都一律可以替換)
2.執行中的 process 需要載入未載入的頁面，因此發生 page fault，但頁框都滿了，所以要執行 page replacement
3.輪到下一個 process 執行，恰好它要用的頁面被換掉了，於是發生 page fault，然後進行 page replacement
4.由於 process 都花時間在 page fault、page replacement 上，所以 CPU 的使用率會降低，於是 CPU 提高 multiprogramming 的 degree，執行更多 process
5.page fault、page replacement 更嚴重→Thrashing 現象

3. 如何處理：

1.局部替換演算法、優先權演算法	如果 process 正發生 thrashing，則此 process 不能再從別的 process 取得 frame，在做 page replacement 時，只能對自己使用到的 page 做置換。這樣可以把猛移現象限制在局部的範圍內。
2.Working Set Model	Working set window 代表 process 最近△次存取的頁面 頁框總需要量 $D=\sum WSS_i$ WSSi：每個 process 的 working set 的 frame 數 若 D>M(實際可用頁框量)，就會出現 thrashing 現象，因此 OS 可以先行防範
3.Page Fault Frequency	OS 設定 page fault ratio 的**上、下限**，若 process 的 page fault ratio 過高，OS 就多分一點頁框給 process，若太低，則搶走一些，須配合**全域放置跟可用頁框池**。

4. **Working set**：

working set window 用來記錄程序最近△時間內的存取頁面

1	2	3	4	5	6	7	2	3	1	2	1	2	1
						t1 ↑						t2 ↑	

以上表為例，時間點 t1 的 working set window 記錄 {1,2,3,4,5,6,7}window size=7，而時間點 t2 則記錄 {1,2,3}window size=3；由此可以推估在某一時間店的 frame 總需求量是多少，若系統的可供給量小於總需求量，表示可能發生 thrashing，此時系統可先暫停某些程序，將其佔用的 frame 釋放出來給其它程序使用。

Window 太大：Degree 降低，系統使用率降低

Window 太小：就會造成 Thrashing

八、Page size 一些考量

Page size 大	優點	1. Page fault ratio 減少
	缺點	1. I/O transfer 時間長 2. internal fragmentation 較嚴重
Page size 小	優點	1. 記憶體使用率高 2. I/O transfer 時間短 3. internal fragmentation 較輕微
	缺點	1. Page Table 大，佔空間 2. Page fault ratio 增加

由於 CPU 越來越快，Memory 容量越來越大，所以 page size 的設計也朝大 size 發展。

九、Locality

1. Temporal Locality(時間局部性)
2. Spatial Locality（空間局部性）
3. Sequential Locality（循序局部性）

Ch10,11 File System

一、檔案基本定義

1. 檔案型態可以是 numeric、character、binary
2. 紀錄 Name、identifier、type、location、size、protection、時間日期
3. 提供操作：Create、Write、Read、Reposition within file、Delete、Rtuncate、**Open(F)**、**Close(F)**
4. **Open-file-table**：系統的表格，紀錄所有開啟檔案的資訊而每一個 Process 有自己的 **per-process table**，其中有一個指向 Open-file table 的指標，此外表中對額每一個檔案通常會有一個 **open count**，刪除時-1 開啟時+1，當刪除=0 時才把它從 table 中移去。
5. **每一個開啟的檔案有下列資訊**

   File Pointer（檔案指標），指向上一次讀寫的位置

   File open count

   Disk location of the file

   Access right

   作業系統對於行程開啟，提供了 lock 的功能，分為 **shared lock** 和 **exclusive lock**。上鎖機制分為**強制(Mandatory)**和**建議(Advisory)**

二、存取方法

1. **循序存取 sequential access**
2. **直接存取 direct access**：使用者提供作業系統相對性的 block 號碼(relative block numbwe)。
3. **用直接存取可模擬循序存取：**

sequential access	direct access
reset	cp = 0
read next	read cp cp = cp + 1
write next	write cp cp = cp + 1

4. **index**：建立在可直接存取的基礎上，例如 IBM 的 **ISAM**

三、目錄結構

1. 可將一個磁碟分割成兩種目錄系統(Partition、IBM 上稱 minidisk)，也可將兩個磁碟合併成為一個目錄系統(Volume)
2. **Single level Directory**
3. **Two Level Directory**：每個使用者擁有自己的檔案目錄
4. **Tree Structure Directory**：注意兩種 Delete 資料夾的方法
5. **Acyclic Graph**：提供共用目錄及檔案的能力，共用有兩種實作方法

   a.Link 實作

   b.Duplicate directory entries ：把相同的目錄複製，不易維持一致性。

   要 Delete 則有三種選擇，

   a.使用者要刪就刪

   b.去找所有 link，把他們都刪掉

   c.保留所有 link，其他 user 存取才知道原來沒了，可能會造成如果新檔案檔名一樣的問題，可用 **timestamp** 解決
6. **General Graph Directory**：cycle 可能會在，可定期用 **Garbage Collection** 的方法來解決
7. **UNIX 的 inode**

Soft link：要指到同一個檔案時自己有一個 inode

Hardlink：直接指到那個共用的 inode，所以要一個 count

四、Mount

五、檔案分享

1. 多使用者：現在系統大多不用個人的管理，而是進化到 Group 的概念或是擁有者存取的概念
2. 遠端檔案系統：演進分為三種

   FTP

   Distributed file system(DFS)

   World Wide Web
3. **Client-Server**
4. **Distributed Information system**：

   DNS：Domain name system

   NIS：Net Work Information system（for UNIX）

   CIFS：common internet file system(CIFS)

   LDAP：lightweight directory-access protocol(For 工業)

六、Consistency Semantics

1. **UNIX semantics**：馬上可以被其他人看見
2. **Session semantics**：已經開啟的就看不到改變了

七、Layered file system

application
logical file system
file organization
basic file system
I/O control
device

八、File system Implement

分成兩種表格

1. **System-wide open file table**
2. **per-process open file table**

3. **目錄製作：可分為 Lined 和 Hash table**
4. **配置方法：Contiguous、Linked、indexed**
5. **Contiguous：**支援 Random access，但要怎麼分配空間？可能可以用 Dynamic storage-allocation、compacts 來解決
6. **Lined：**一個 block 串一個這樣接，但存取太花時間且一個壞掉就全部壞掉了，可以用 **cluster**（多個 cblock 視為一個單位）改善
7. **indexed：**問題是一個索引的 block 要多大？想越短越好，有一種改善方法

如果三個要存取一個在檔案中位置 k 的檔案：

Contiguous 要 1 次、Lined 要 k 次、indexed 要 2 次

8. 可用空間管理：用 **Bit Vector、Linked List（可用 FAT 改善）、Group**(把連續可用的 n 個位址放在第 1 個 block 中，行程一個群組，最後一個在指向另外一個可以用的群組)、**Counting**（紀錄這個 block 後面接了幾個可用 block）
9. **log-based transaction-oriented or journal**

10. **一致性緩衝區**

九、Virtual File system

1. **VFS(virtual file system)：**用來轉換各種不同的 file system

十、NFS

1. 要先 **mount**，透過 **mount protocal**，是**非透明**的方式（因為要知道對方的位址）
2. NFS protocol：提供以下運作

   Searching for a file within a operations

   Reading a set of directory entries

   Manipulating links and directories

   Accessing file attributes

   Reading and write files

因為 NFS 是 stateless，所以沒有 **open()**和 **close()**，因為送出去的訊息都要全部講，包括要操作的檔案。

Ch12,13 Storage and I/O System

一、磁碟一些名詞

1. **Disk arm**
2. **track**
3. **cylinder**
4. **sector**

二、存取時間

1. **Seek time**
2. **rotation latency**
3. **transfer time**

三、磁碟連結

1. 典型桌上型電腦透過 **IDE** 或是 **ATA**，更高階的工作佔用 **SCSI** 或是光纖 **FC** fiber channel
2. **NAS(Network-attached storage)**和 **LAN WAN**、**SAN(Storage area network)**的關係

四、磁碟排班

1. **FCFS**
2. **SSTF(shortest seek time first)**
3. **SCAN**
4. **C-SCAN**(有滑過去的)
5. **LOOK**
6. **C-LOOK**

五、RAID

Ch14,15,16 Distributed Operating System

一、分散式系統有以下四優點

1. **Resource Sharing**
2. **Computing speedup**
3. **Reliability**
4. **Communication**

二、網路(Network)作業系統有下列型態

1. **Remote logging**
2. **Remote Desktop**
3. **Transferring data 如 FTP（非透明）**

三、分散式作業系統

1. **Data Migration (轉移)**
2. **Computation Migration**
3. **Process Migration**

   Process Migration 基於以下理由：

   Load balancing

   Computation speedup

   Hardware preference

   Software preference

   Data access

四、網路型態

1. **LANs(Local-area networks)**
2. **WANs(Wide-area networks)**

五、網路拓樸架構

有一些要考量：**安裝代價、通信代價、可靠度**

六、通信結構

1. **Naming and name resolution**：如何把人看的名字轉換成數字讓機器知道兩種方法，放一些資訊在電腦裡，或是透過 **DNS**
2. Routing strategies：可分為 **fix routing**、**virtual routing**（路徑固定的，但不同時間內 A 傳 B 路徑可能不同）、**dynamic routing**（選一條來用），固定和動態可能合併，主機由固定的聯道 **gateway**，再動態連到其他網路，路徑上則有一個 **router** 負責安排
3. **Pocket strategies**：看要不要告知已經到目的地了
4. **Connection strategies**

   Circuit switching：兩行程要聯繫，經由固定線路

   Message switching：暫時性的連結

   Packet switching：一個訊息分為很多封包，且每個封包可能都經由不同路徑到達
5. **Contention（衝突）**

   衝突偵測(CSMA/CD)：有空就傳，有衝突就 CD，項乙太網路就是

   token passing

六、通信協定

1. **標準的架構**

應用層 (application layer	和使用者連接	HTTP、DNS、FTP、telnet
表現層 (presentation layer)	負責解決網路中不同站上所使用的格式，以及半雙向全雙向的轉換	
會議層 (session layer)	付則實現 session、同步等控制	
傳遞層 (transport)	負責網路的低階存取和使用者之間的溝通，包括將訊息分割為封包	TCP-UDP
網路層 (network)	負責網路的通信以及傳送，包括封包送出的位址控制	IP
資料連結層 (data-link layer)	負責處李固定長度的封包或 frames，包括偵測修復	
實體層 (physical layer	傳遞一串位元或去，都要用 01 的方式	

註：**TCP**(transmission control protocol)

UDP(user datagram protocol)

七、強韌性 Robustness

1. **故障偵測：用 handshaking**(互傳 **I am up**，突然有一個沒傳到的話考慮要等下一次該傳的時間，還是直接問 **Are you up**?)
2. **重組架構：**連結線壞掉，系統要告訴大家更新路徑表，不要在走了。點壞掉，系統要告訴大家不要去存取他的東西
3. **故障修復：**線修復，要通知連結到線的點，點修復，系統告訴大家可以存取了。

八、設計議題

1. 讓使用者感到透明
2. 容錯
3. 規模性(scalability)
4. 容錯和規模和規模性是有關聯的，過度負載就會造成癱瘓，所以，備而不用市容錯很重要的關鍵，分散式系統正好具備這樣的潛力。
5. 集中式控制以及集中資源不應該用來建造一個規模性以及容錯的系統，但又常用到，例如沒有硬碟的系統，這時候 clustering 就可以用上了。

九、網路的例子（很多名詞）

1. TCP/IP 網路上，每台主機都有一個相關的 32 位元號碼，分為網路號碼和主機號碼，網路號碼由 Internet 管理者決定後主機號碼就可以自己選了
2. 傳送封包到 TCP/IP 層，這時候要怎麼在網路中移動？
3. 每一台乙太網路都有一個位元組號碼，叫做 **MAC**(medium access control address)，**LAN 上兩台電腦要透過這個號碼才能互相通訊**，如果兩台電腦要連結，核新會產生包含目的系統 IP 位址的 **ARP** 封包（address resolution protocol），系統中符合相同位址的主機會回覆，並回傳 MAC 位址。
4. 這樣很麻煩，因此主機內會有一個 **IP-MAC** 的 Cache，並且有 aging。
5. 一台乙太網路宣布他的識別碼和位址之後就可以開始通信，這時候以傳封包的方式開始，如果在相同的區域網路就找出目的主機的位址，並且放封包在電線上，目的地看到是自己以後，就開始讀封包。
6. 如果是在另外的區域網路，則經由 router 沿著 **WAN** 傳送。

以下為分散式檔案系統

十、基本定義

1. DFS 的檔案系統中，用戶，伺服器以及儲存裝置散佈再系統各機器之間，特色就是系統中用戶以及伺服器的多重性以及自主性。
2. 理想情況下，DFS 應該被用戶是唯一個傳統的檔案系統，因此應該是透明(transparent)的。
3. 最小的單位是 **component unit**

十一、命名與透明性

1. **Location transparency**：地點透明性，就是用戶不會知道檔案實體儲存位址。
2. **Location independence**：檔案實體儲存為地點改變，該檔案名稱不需要改變，因此適合用在檔案轉移，他可以在不同時間把不同檔案對應到不同的位址。
3. 用戶可能是 diskless，因此存取呼叫 ROM 內的程式，這樣的方式可以節省成本以及獲得便利（例如系統升級的時候只要升級一台）
4. 命名方法有三種：
   a.以 **host:local-name** 來區分：非透明性
   b.以 **mount** 的方式存取，早期要知道先知道目錄安裝好才能用，現在發展到 automount 了
   c.完全整合，以獨立的全體性命名來命名分散式喜桶內所有檔案
5. 實作上，檔案名稱與地點對映，先把他們集合成一個 component unit，引用 **location indepent file idenfiers** 地點獨立檔案識別字，先標明檔案所屬的單元，再標明單元內的獨特檔案。

十二、遠程檔案存取與一致性

1. 用 RPC 來呼叫
2. 為了更好的性能，通常會用 cache 的方式，但不提供磁碟快取。但 NFS 提供了 cachefs，這是用戶端的磁碟快取。
3. 三種策略
   write throught policy
   delayed write policy：每隔一段時間（如 30 秒）寫入
   write on close policy
4. 如何維持一致性？
   client-initiated approach
   server-initiated approach

十三、stateful and stateless

1. **Stateful**：如果伺服器壞了，那要透過 **orphan detection and elimination** 來回復，將造成 sever 中斷執行。
2. **Stateless**：代價是要求較長的訊息，不過伺服器壞時修復容易。
3. UNIX 隱含差距值的做法，天生就具有狀態，伺服器必須要有對應到索引點的表格，並把目前的 offset 存入一個檔案，因此在 UNIX 上實做 NFS 不用把檔案描述值 offset 等資訊也放在指令內。

Ch17,18 Protection and security

一、基本定義、原則

1. Protection 主要對於系統內多使用者環境，而 Security 則對於網路上多個系統
2. 保護原則：**Principle of least privilege**，最少特權

二、存取矩陣(access matrix)

※另外注意一下 **confinent problem**：可以存取權力傳播，但不能防止資訊的接漏到執行環境外