作業系統第一章重點整理

一、何謂作業系統與核心

OS = 軟體與韌體組成，驅動硬體，提供使用者介面與資源管理。
核心（Kernel）：OS 的核心部分，負責低層資源控制。
目的：讓使用者好用、讓硬體高效率。
名詞：韌體（Firmware，燒錄於 ROM/EPROM 等）。
易錯點：Shell 並非 Kernel；Shell 是介面與命令解析。

二、作業系統四大管理

記憶體管理：配置/回收/置換，決定常駐與換出。
處理器/處理程序管理：程序狀態與排程（如 Ready/Running/Blocked），決定 CPU 使用時機與時間長度。
設備管理：通過 Device Driver 控制 I/O 裝置，處理共享/專屬設備協調。
資訊（檔案）管理：檔案/目錄結構、存取控制、磁碟配置。
典型考點：四大管理「對應哪些資源」、與「各自的任務」。

三、開機載入原理（Bootstrap Loader）

上電→PC 指向啟動載入器（韌體）→載入 OS Kernel 至主記憶體→PC 指向 Kernel→控制權交給 OS。
命令直譯器通常由核心包含或隨核心啟動，負責接收與解析命令。
考點：啟動載入器角色、PC 流程、核心取得控制權的時刻。

四、命令直譯器（Shell）

功能：解析命令、檢查正確性、呼叫對應程式。
內部命令：開機與外殼一起載入、常駐；外部命令：存於磁碟、需時才載入。
批次處理：將多條命令寫入檔案，一次性執行。
易混淆：Shell ≠ Kernel；CLI/GUI 都可以是介面的一環。
批次處理（Batch Processing）
特色：收集後一次執行、互動性低、非即時、周邊 I/O 時 CPU 常閒置、往返時間長。
適用情境：開機環境設定、學生成績彙算、稅務批次計算、例行報表等。
考點：與分時/即時的差異、Turnaround Time 定義。

五、多重程式（Multiprogramming）

核心概念：當前程式 I/O 等待時，CPU 切給其他程式使用。
效益：提升 CPU utilization、throughput；讓周邊也盡量忙碌。
與分時關係：分時必須建立在多重程式能同時常駐的前提上。
考點：何以提升產量與使用率；與分時的差異性。

六、分時系統（Time Sharing）

時間片（ms 級）輪轉分配；一片到期則切換；支援互動。
機制：CPU Scheduler 從 Ready Queue 取任務；I/O 或等待事件時移至 Blocked Queue；Reschedule 後確保公平。
關係命題：為達快速輪服務，分時一定提供多重程式；但多重程式不一定提供分時。
考點：時間片概念、公平性、互動性、與多重程式的邏輯關係。

七、即時系統（Real-Time）

重點：在限定時效內正確反應；Response Time 關鍵。
分類：
硬即時（Hard RT）：必須於期限內完成所有處理。
軟即時（Soft RT）：期限內要回應，但可持續佔用 CPU 做善後。
與 Online 關係：即時系統必須在線；但在線系統不一定即時。
考點：Hard vs Soft 定義、Response Time、Online 與 Real-Time 的區分。

八、週邊設備線上同時工作（Spooling）

目的：把專屬裝置（如印表機）「模擬成共享」，避免輸出交錯。
作法：Spooler 先將各程式輸出寫入硬碟檔，再按序送至裝置。
效益：提升併發度與整體吞吐，避免「你一行我一行」問題。
考點：Spooling 與快取/緩衝的差異、為何能「共享」專屬裝置。

九、多處理器與多核心

多處理器系統：一台電腦多顆 CPU/多核心，共用匯流排、記憶體、周邊、同一 OS；別名平行系統/多重處理。
效益：效能與可靠度提升；單顆壞不致全機停擺，但效能下降。
多核心 CPU：單晶片多核心；超執行緒（HT）在單核心邏輯模擬兩邏輯處理器以並行執行執行緒。
考點：多處理器 vs 多核心 vs 超執行緒的概念差異與效益。

十、分散式系統（Distributed Systems）

透過通訊線路連接多台電腦以分工、交換訊息、共享資源。

優點：資源共享（提升可用性）、計算加速與負載平衡、可靠度、通訊（如 Email/FTP）。

考點：與多處理器系統的差異（單機多 CPU vs 多機透網連結）。

第一章高頻考點速記

1. 四大管理：記憶體、處理器/處理程序、設備、資訊（檔案）。
2. 分時 vs 多重程式：分時一定包含多重程式；多重程式不必然分時。
3. 硬即時 vs 軟即時：硬即時「必須按時完成全部工作」；軟即時「按時回應，但可持續善後」。
4. Spooling 定義與目的：以磁碟暫存輸出，將專屬裝置「共享化」，避免交錯輸出。
5. Bootstrap Loader 流程：Power on → PC 指向啟動載入器 → 載入 Kernel → PC 指向 Kernel → OS 接管。

作業系統第二章重點整理

一、現代電腦的運作與控制器

流程：程式與資料自輔助記憶體（磁碟）搬至主記憶體→CPU 取指（PC 指向下一指令）→資料送入暫存器計算。

控制器：Disk Controller（控硬碟）、Interface Card（控 I/O 裝置）、Memory Controller（仲裁主記憶體存取與同步）。

併行：CPU、磁碟、I/O 裝置可並行；但主記憶體單時刻單用，需同步。

考點：為何需要 Memory Controller 同步？因主記憶體單埠/共享資源。

二、輪詢（Polling）與中斷（Interrupt）

輪詢：主控單元按序查詢多個從屬單元狀態；浪費 CPU；在多設備時用於確認中斷來源。

中斷：設備/程式觸發；CPU 停下現行工作，OS 接管；中斷遮罩暫存器（IMR）可致能/屏蔽特定 IRQ。

比較：中斷效率高；但多設備卡時常需搭配輪詢以確定來源。

陷阱：軟體中斷（錯誤或系統呼叫）；系統呼叫屬「可預期陷阱」。

考點：IMR 的作用、Trap 與硬體中斷差異、系統呼叫為何屬陷阱。

三、中斷處理與系統呼叫步驟

中斷流程：OS 控制→保存狀態（暫存器, PC, Stack）→判斷類型、呼叫 ISR →恢復狀態→繼續執行。

優先等級：高優先可嵌套打斷低優先；即時系統（RT）設定高優先以保證時限。

系統呼叫：User Mode 發起 Trap→切到 Kernel Mode→執行 Service Routine →返回 User Program。

考點：保存/恢復哪些狀態、嵌套中斷的時機與風險、User/Kernel 模式切換。

四、DMA 與週期盜取（Cycle Stealing）

DMA 目的：大量高速 I/O 搬運，減少中斷次數、提升 I/O 產量。

DMA 行為：CPU 可在 DMA 期間閒置；DMA 信號不同於一般中斷；OS 不必每次保存/恢復狀態。

週期盜取：I/O 裝置在記憶體一個存取週期中「偷用」匯流排與主記憶體；合成多次週期盜取可完成 DMA。

考點：DMA vs 中斷差異（資料傳輸路徑、負擔、延遲）、何時選用 DMA（高速大量）、週期盜取對 CPU 的影響。

五、緩衝區、字元/區塊設備、Blocking

緩衝區：位於 CPU—裝置或主記憶體—介面—裝置之間；平衡速度差。

字元設備：鍵盤/滑鼠，單位為字元；區塊設備：磁碟，單位為區塊。

實體記錄（Physical Record）=區塊=磁區；邏輯記錄（Logical Record）=使用者的一筆資料。

Blocking：多筆邏輯記錄合併為一個實體記錄，減少 I/O 次數，提高效率。

考點：為何 Blocking 能提升效率？因每次 I/O 成本高，合併可降低總次數。

六、階層式記憶體與 Von Neumann

層級：Register（最快/最貴/最小）→Cache→Main Memory→Secondary Memory（最慢/最便宜/最大）。

揮發性：主記憶體、快取、暫存器屬揮發；輔助記憶體不揮發。

OS 角色：適當的快取大小＋置換策略可達 80–99% 命中率。

Von Neumann：所有指令與資料在執行前需進主記憶體；系統總線連接 CPU、記憶體與 I/O 控制器；存在中斷與 DMA 通路。

考點：層級的速度/成本/容量趨勢；命中率與置換策略；Von Neumann 核心觀念。

七、系統保護

雙模保護：User Mode（受限指令/資源）vs Supervisor/Kernel Mode（特權指令/資源）。

登入保護：身份驗證與授權。

四大管理保護：記憶體（存取控制/界限檢查）、處理器（時間/特權）、設備（I/O 權限與驅動防護）、資訊（檔案權限與 ACL）。

考點：為何需要雙模式？阻隔使用者程式直接動到硬體/特權；OS 如何配合硬體達成保護。

高頻考點速記

1. 輪詢 vs 中斷：中斷效率高；多設備情況下仍需輪詢確認來源。
2. 陷阱與系統呼叫：皆屬軟體中斷；系統呼叫為「預期」的陷阱。
3. 中斷處理五步：接管→保存→判斷/ISR→恢復→續行。
4. 優先等級：高優先可嵌套打斷；即時系統優先級最高。
5. DMA 適用：大量高速 I/O；週期盜取延遲=一個記憶體週期。
6. 緩衝/Blocking：合併邏輯記錄成實體記錄，降低 I/O 次數。
7. 記憶體階層趨勢：越上層越快/貴/小；OS 置換策略提升命中率。
8. 雙模保護：使用者模式受限、核心模式特權；配合登入與資源權限。

作業系統第三章重點整理

一、何謂處理程序

定義：正在執行中的程式（program in execution）
元素：程式碼、執行中的資料與上下文、開啟的資源（檔案、I/O）
與程式的差異：程式是靜態檔案；程序是動態執行個體
考點：Process vs Program、上下文內容包含哪些

二、程序控制塊（PCB）

內容：PID、程序狀態、程式計數器（PC）、暫存器集合、排程資訊（優先權、時間片）、記憶體描述（頁表/段表）、I/O 狀態（開啟檔案、裝置）、帳務資訊（CPU 時間等）
作用：支撐上下文切換與排程、故障回復、資源管理
考點：PCB 欄位、上下文切換步驟

三、程序記憶體佈局

Text（程式碼）、Data（已初始化/未初始化 BSS）、Heap（動態配置）、Stack（函式呼叫、區域變數、返回位址）
穩定性：堆向上成長、堆疊向下成長（常見示意）
考點：各區段功能與常見錯誤（堆疊溢位、記憶體洩漏）

四、程序狀態與佇列

狀態：new → ready → running → blocked(waiting) → ready（I/O 完成） → terminated
佇列：Ready Queue、Device Queue（以裝置分類）
上下文切換：保存暫存器/PC/狀態 → 切入新程序 → 恢復
考點：何時進入 blocked（I/O/同步等待）、就緒與執行差異、上下文切換開銷

五、程序生成與載入

UNIX：fork() 產生子程序（幾乎完整複製父程序映像與描述元），子程序通常緊接 exec() 載入新程式（取代映像），父子以返回值區分分支
Windows：CreateProcess() 指定可執行檔與環境，一次建立/載入
環境與繼承：子程序繼承檔案描述元、工作目錄、環境變數、權限
考點：fork/exec 角色分工、返回值判斷父子分支

六、程序終止與等待

exit(status)：釋放資源、回傳狀態碼給父程序

wait()/waitpid()：父程序同步等待特定子程序；避免殭屍

非正常終止：signal（如 SIGKILL）、abort

殭屍/孤兒處理：殭屍保留最小 PCB/表格；孤兒由 init/system process 接管

考點：殭屍形成原因與解法、孤兒的接管機制

七、系統呼叫與 I/O 程序

使用者模式呼叫 write/read → trap 進核心模式 → 檢查參數/權限 → 驅動/控制器執行 I/O → 中斷回報 → 內核喚醒等待的程序 → 返回使用者模式

I/O 程序與主程序：主程序可能因 I/O 進入 blocked；OS 排程其他就緒程序執行

考點：阻塞式與非阻塞式 I/O、同步/非同步 I/O 差異

八、程序間關係與通訊（簡述）

父子樹：以 PID/PPID 形成樹狀結構；可使用 wait 進行同步

IPC 概覽：管道（pipe）、訊息佇列、共享記憶體、信號（signal）、socket

考點：基本 IPC 適用情境與同步需求（詳見 IPC 章）

九、排程概述（與程序相關）

常見演算法：FCFS、RR（時間片）、SJF/Shortest-Remaining-Time、優先權（含老化機制）

指標：CPU 使用率、吞吐量、回應時間、等待時間、周轉時間

考點：時間片對互動性的影響、SJF 對平均等待時間的最優性

高頻考點速記

1. Process＝program in execution；PCB 保存上下文與資源
2. 狀態五步：new→ready→running→blocked→terminated
3. fork→exec：先複製再載入；Windows 用 CreateProcess 一步到位
4. 殭屍＝子結束父未 wait；孤兒＝父先結束由 init 接管
5. 上下文切換：存暫存器/PC→切入→恢復；有開銷
6. I/O 呼叫：trap 轉核心→驅動工作→中斷回報→喚醒程序

作業系統第四章重點整理

一、使用處理程序的缺點

三個 Word「用程序寫」：主記憶體內出現三份地址空間 → 三份 Code/Data/Resource/PC/Regs/Stack

成本與資源：記憶體佔用高、切換昂貴、共享資料需 IPC

考點：為何多程序造成三份程式碼與資料段？因為位址空間互相隔離

二、主記憶體使用方式

程序版：N 程序 → N 份位址空間

執行緒版：1 程序位址空間中有 K 執行緒（共用 Code/Data/Resource；各自 PC/Regs/Stack）

考點：畫出對照圖，標註共用與私有部分

三、資料使用方式

多程序：必須透過 IPC（pipe、共享記憶體、訊息佇列、socket）互傳資料/資源

多執行緒：在同一位址空間內直接共享資料

風險：多執行緒共享容易產生競態條件（需同步）

考點：為何執行緒共享快但危險？因為缺乏隔離，需同步機制保護

四、建立與終止

程序派生：每啟動一次 Word → OS fork 一個新程序；結束時 OS 處理 exit（清理資源）

執行緒派生：OS 建立首個程序後，後續 Word 功能可在同位址空間內新增執行緒；最後一個執行緒結束再做程序的 exit

考點：程序與執行緒的「建立/終止」誰比較昂貴、為何？

五、環境切換（Context Switch）

程序切換：要切換/複寫位址空間對映（TLB 失效、頁表切換）、另加保存/恢復暫存器、PC、堆疊指標 → 成本高

執行緒切換：同一程序內僅切 PC/Regs/Stack → 成本顯著較低

考點：TLB 與頁表對程序切換成本的影響；為何執行緒切換較輕

六、何謂執行緒（定義匯總）

程序之中的輕量執行單元（LWP），共用地址空間與資源

私有執行上下文：PC、暫存器、堆疊；共享執行內容：Code/Data/資源

應用：GUI 前景互動＋背景 I/O、伺服器的多連線處理

考點：列舉共享與私有項目；何時適合用多執行緒

七、常見延伸（若後續頁面涵蓋，這些是高頻考點）

同步原語：Mutex、Semaphore、Condition Variable、RWLock

問題典型：Race Condition、Deadlock、Livelock、Starvation

設計模式：Thread Pool、Producer-Consumer、Pipeline

考點：為何多執行緒須同步？如何避免競態與死結？

高頻考點與易混提醒

1.多程序 vs 多執行緒：隔離性 vs 共享性；IPC vs 直接共享；切換昂貴 vs 輕量

2.共用/私有清單（必背）：

共用：Code / Data / Resource（檔案、開啟的 Handle 等）

私有：PC / Registers / Stack

3.切換成本來源：程序切換涉及位址空間與 TLB；執行緒切換不動位址空間

4.共享風險：多執行緒需同步，否則競態/資料毀損

5.何時選用多程序？安全隔離、高可靠性；何時選多執行緒？高共享與高互動、低切換延遲

作業系統第五章重點整理

一、排程目標與效能指標

系統目標（依情境）

批次系統：最大吞吐、最小平均等待/周轉時間

互動系統：最小回應時間、平順體驗、避免飢餓

即時系統：期限（Deadline）滿足率、抖動控制

效能指標

CPU 使用率（Utilization）

吞吐量（Throughput）：單位時間完成數

回應時間（Response Time）：提交到首次回應

等待時間（Waiting Time）：於就緒佇列中等待總時長

周轉時間（Turnaround Time）：提交到完成總時長

二、工作負載特性與到達模型

程序行為：CPU burst 與 I/O burst 交替

到達分佈：同時到達 vs 不同到達時間（決定甘特圖與計算）

估計 CPU burst：指數平滑 $\tau n+1 = \alpha \cdot tn + (1-\alpha) \cdot \tau n$（$\alpha$ 越大越看重最新樣本）

三、單 CPU 排程演算法

FCFS（先來先服務）

優點：簡單、低開銷

缺點：護航效應（長工作阻塞短工作）；平均等待可能高

適用：磁帶/批次、上下文切換成本高

SJF（最短工作優先）/ SRTF（最短剩餘時間優先，搶先）

理論：對同分佈 CPU burst，SJF 使平均等待最小；SRTF 是其搶先版

前提：需預測 CPU burst（用歷史估計）

風險：長工作飢餓；需「老化」或多層回饋緩解

Priority（優先權排程，分為搶先/非搶先）

來源：靜態（類型/使用者）或動態（等待時間、I/O 比）

問題：低優先可能飢餓 → 老化（aging）逐步提升優先

RR（時間片輪轉）

互動最佳化：小 q 改善回應時間

取捨：q 太小→上下文切換開銷↑；q 太大→退化為 FCFS

經驗：q 約 10–100ms 視系統與負載

多層佇列（Multilevel Queue, MLQ）
依工作類型分層：前景（互動）/背景（批次）等，各層自有策略
層間排程可固定優先或時間比例
多層回饋佇列（Multilevel Feedback Queue, MLFQ）
規則：新進程放高層；用完整片未完成→降層；長期等待可升層（老化）
目標：促進短作業優先與互動性、公平性兼顧
參數：層數、各層 q、升降規則（考題常問調參效果）

四、計算與範例方法（必會）
畫甘特圖：依到達/執行/切換時刻標出
等待時間：完成執行外的排隊總和
周轉時間：完成時刻 − 到達時刻
平均值：對所有進程平均
常見陷阱：不同到達時間、搶先情況、I/O 阻塞導致就緒佇列變化

五、上下文切換與開銷
內容：暫存器、PC、堆疊等保存/恢復
影響：切換頻率↑ → CPU 有效工作時間↓
RR/MLFQ 的 q 需與切換成本平衡

六、多處理器/多核心排程
負載平衡（Load Balancing）：推送/拉取策略
處理器親和性（Processor Affinity）
軟親和：傾向回到原核心
硬親和：限制只在特定核心執行
好處：Cache 命中↑、NUMA 記憶體區域性↑
全域佇列 vs 每核佇列：全域易平衡、每核減少鎖競爭
鎖競爭與臨界區：多核下排程器自身也需鎖設計（如 RCU、無鎖結構）

七、即時排程（簡要）
任務屬性：週期性（C, T, D）與非週期性
RM（Rate Monotonic，固定優先）：週期越短優先越高；利用率上限 m 任務 U <= m(2^(1/m) − 1)
EDF（Earliest Deadline First，動態優先）：最早期限先排；可達最優可行排程（單機理想條件）

八、工程實務與策略搭配

互動工作：RR/MLFQ，q 動態調整
混合工作：MLFQ 或 CFS（完全公平調度理念，按虛擬執行時間）
防飢餓：老化、回饋升層
優先反轉（Priority Inversion）：以優先權繼承/上限協定緩解

九、高頻考點與易錯整理

SJF/SRTF 理論最小平均等待；但需預測 burst
RR 的 q 對回應性與開銷的影響必考
Priority 需老化；MLFQ 調參理解（層數/時間片/升降規則）
護航效應：FCFS 長工作在前拖累短工作
即時：RM 固定優先 vs EDF 動態最優；利用率上限公式
多核：親和性與負載平衡取捨，全域 vs 每核佇列的鎖競爭差異

速記口訣

短者先排：SJF/SRTF
片小快、片大慢：RR 的 q 取捨
老化防餓：Priority/MLFQ
親和回原核、快取更友善
RM 看週期、EDF 看期限

作業系統第八章重點整理

一、核心概念與名詞

死結（Deadlock）：一組進程彼此等待對方持有的資源而永不前進
飢餓（Starvation）：個別進程長期得不到所需資源（不一定構成死結）
活鎖（Livelock）：進程不斷改變狀態以避免衝突，但實際上不前進
資源類型：
可剝奪資源：可被系統強制收回（如 CPU）
不可剝奪資源：不能被強制收回（如印表機）
可重用資源：使用後可重複（記憶體頁框、I/O 裝置）
可消耗資源：使用後消失（訊息、工作單位）

二、死結成立的四必要條件（必要但非充分）

互斥（Mutual Exclusion）：資源一次只能被一進程使用
保持並等待（Hold and Wait）：進程持有資源同時等待其他資源
不可剝奪（No Preemption）：資源不可被強制收回
循環等待（Circular Wait）：形成資源請求的環形鏈 提示：破壞任一條件 → 死結不可能成立

三、處理策略總覽

預防（Prevention）：在設計上破壞四必要條件
破互斥：以分割/複製等方式讓資源可共享（通常困難）
破保持並等待：一次性請足（可能造成資源利用率低）
破不可剝奪：允許剝奪（對不可剝奪資源不易）
破循環等待：對資源進行「線性排序」，只按序上升請求
避免（Avoidance）：在執行時判斷是否會進入不安全狀態，拒絕可能導致死結的配置
銀行家演算法（Banker’s Algorithm）
檢測與恢復（Detection & Recovery）：允許死結發生，定期檢測並採取恢復措施
檢測：資源分配圖（RAG）或矩陣檢測演算法
恢復：剝奪資源、回滾（Checkpoint/Restart）、終止進程

四、銀行家演算法（重點）

目的：確保系統始終可找到「安全序列（Safe Sequence）」

資料結構：

Available：目前可用各資源數量向量

Max：每進程對各資源的最大需求矩陣

Allocation：已分配給各進程的資源矩陣

Need：尚需矩陣，Need = Max − Allocation

安全性檢查流程（判斷當前是否安全）：

Work = Available；Finish[i] = false

找到某 P i 使 Need[i] ≤ Work（逐資源向量比較）

若找到，Work += Allocation[i]；Finish[i] = true；加入安全序列；回到步驟 2

若不再有符合者，檢查所有 Finish 是否為 true；是→安全；否→不安全

資源請求判斷（Request[i]）：

若 Request[i] ≤ Need[i] 且 Request[i] ≤ Available，暫時「試配」給 P i 重新做安全性檢查；若仍安全→正式分配；不安全→拒絕請求

關鍵理解：

不安全狀態 ≠ 必定死結，但可能走向死結

核心是能否找到一條完成所有進程的安全序列

五、資源分配圖（RAG）與檢測

節點：進程節點 P、資源節點 R（每種資源可有多個實例）

邊：

請求邊（P → R）：進程請求資源

分配邊（R → P）：資源分配給進程

單一實例情況：若圖中存在「有向環」，則系統必死結

多實例情況：需要矩陣演算法；有環不一定死結，但為風險指標

六、恢復策略與選擇

剝奪資源：從某進程回收資源，風險是需要回滾或重新計算

回滾（Rollback）：依檢查點還原進程到安全狀態

終止進程：選擇代價最小者（考量：優先權、已完成工作量、持有資源量、影響範圍）

實務衡量：恢復成本、系統可用性、使用者影響

七、降低死結風險的工程實務

縮短持有時間（盡快釋放）
一次性請求或分段有序請求
資源線性排序（全域鎖序）
設計「超時（Timeout）」與「嘗試鎖（Try-Lock）」機制
使用層級鎖與死結探測器

八、常見考點陷阱與解題技巧

四必要條件：任一被破壞 → 不會死結（但仍可能有飢餓）
Need 計算：必背 Need = Max − Allocation
安全序列：步驟要熟；Work 逐步累加 Allocation 的觀念
不安全 vs 死結：區分清楚；不安全僅代表可能無法完成所有進程
RAG：單實例的有向環 → 必死結；多實例需矩陣檢測
恢復選擇：題目常以「剝奪 vs 回滾 vs 終止」做代價比較
互斥鎖 vs 讀寫鎖：讀多寫少場景用 RWLock 降低競爭，但依舊需注意死結

九、與同步原語的關係（延伸）

Mutex/Semaphore/Condition Variable：用於互斥與同步，設計不當易導致死結
破環技巧：固定鎖取得順序、設定鎖層級、使用 Try-Lock + 超時

速記口訣

四要：互、保、不、循（互斥；保持並等待；不可剝奪；循環等待）
銀行家：三表一向量（Alloc/Max/Need＋Avail）；先找 Need ≤ Work，累加成安全序列
不安全不必死；檢測與恢復要分清
破環靠排序；縮持有時間保安全