第六章

1. preemptive 和 Non-preemptive 的做法 以及兩者的不同?

- 可搶佔式（Preemptive）

  在可搶佔式調度中，操作系統允許當前正在執行的進程在任何時候被強制停止，無論它是否處於用戶模式或內核模式。當一個更高優先級的進程需要運行時，操作系統會中斷當前進程的執行，並將 CPU 資源分配給新的進程。

  (1) 常見演算法：

  - Round Robin（RR）：所有進程依序執行固定的時間片 。
  - Priority Scheduling：高優先級的進程搶佔低優先級進程。
  - Shortest Remaining Time First（SRTF）：執行剩餘時間最短的進程。

  (2) 優點：

  - 能確保高優先級的進程及時執行。
  - 提高系統的資源利用率和響應速度。
  - 適合即時系統。

  (3) 缺點：

  - 需要頻繁的進程切換，可能導致上下文切換開銷增加。
  - 可能導致某些低優先級進程長時間得不到執行（飢餓現象）。

- 不可搶佔式（Non-preemptive）調度

  在不可搶佔式調度中，當一個進程開始執行後，它會一直運行，直到發生進程主動退出內核模式，返回用戶模式、進程自願放棄 CPU，或者主動進入阻塞狀態（例如等待 I/O 完成）、進程執行完畢。

  (1) 常見演算法：

  - First Come First Serve（FCFS）：按照進程到達順序執行。
  - Shortest Job First（SJF）：執行所需時間最短的進程。
  - Priority Scheduling：按照優先級，但不能搶佔當前正在執行的進程。

  (2) 優點：

  - 無需頻繁進行進程切換，上下文切換開銷較低。
  - 邏輯簡單，易於實現。

  (3) 缺點：

  - 可能導致系統資源利用率低，因為當進程執行阻塞操作時，CPU 會閒置。
  - 無法保證高優先級進程能快速執行，不適合即時系統。
  - 可能出現長時間的進程等待現象（例如：短進程等待長進程完成）。

- 區別

特點	可搶佔式（Preemptive）	不可搶佔式（Non-preemptive）
進程切換	操作系統可以強制中斷進程並切換 CPU 控制權。	進程必須自願放棄 CPU 或終止，操作系統不會強制切換。
響應速度	可以更快速地響應高優先級進程的需求。	響應速度較慢，必須等進程完成當前任務或自願讓出 CPU。
上下文切換	需要頻繁的上下文切換，會增加系統開銷。	上下文切換較少，開銷相對較小。
競爭條件	可能會出現競爭條件，特別是在內核模式下。	不會有競爭條件，因為進程不會被中斷。
適用場景	現代多任務操作系統，如 Linux、Windows。	早期操作系統或某些嵌入式系統，要求簡單和穩定的內核設計。

2. 如何運用 compare and swap 去讓我的 schedule 更順

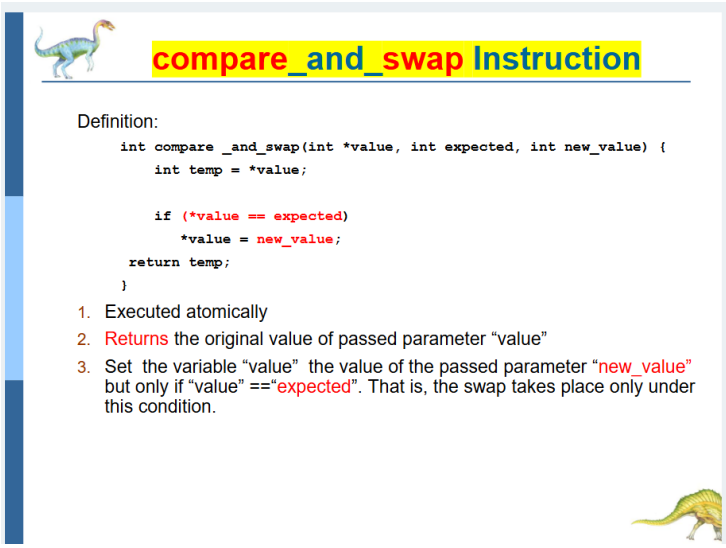
使用自旋鎖來避免傳統鎖帶來的性能開銷。自旋鎖會讓等待的進程不斷地重試 CAS 操作，直到成功獲得鎖。

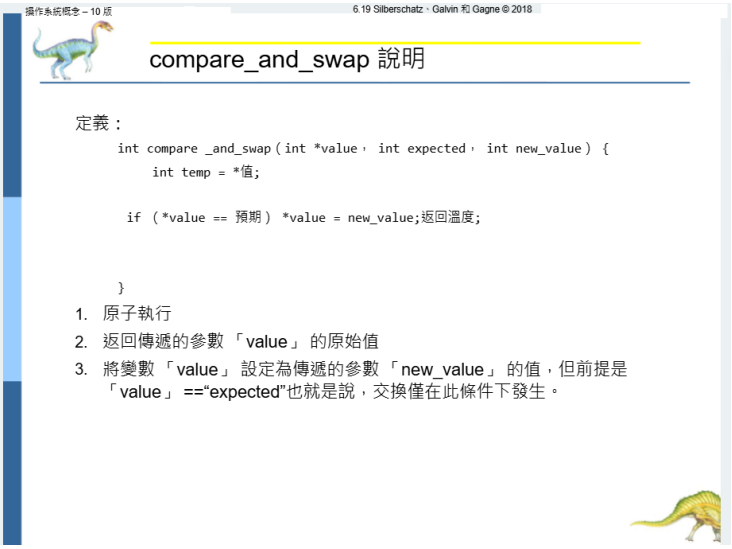
共享整數變數 **“lock”** 初始化為 0。

在排程系統中，CAS 可以用來解決以下問題：

1. 減少鎖的使用：避免傳統鎖機制導致的死鎖與效能問題。
2. 資源分配：確保多執行緒在競爭資源時，只有一個執行緒能成功修改狀態。
3. 任務調度：讓任務隊列能被多個執行緒安全地存取。

```
do {
  while (compare_and_swap(&lock, 0, 1) != 0) ;
  lock = 0 ;
  }
while (true);
```





如何運用 Compare and Swap (CAS) 讓排程更順暢

1. CAS 的基本概念與作用

- Compare and Swap (CAS) 是一種原子操作，主要用於多執行緒環境中解決共享資源競爭問題。
- 它的原理是比較一個變數的當前值（current）是否等於預期值（expected）。如果相等，則交換成新值（new_value），否則保持原狀。
- CAS 操作是無鎖的，因此不會導致執行緒阻塞，能有效減少排程中因鎖引起的等待和死鎖問題。

2. CAS 如何優化排程

1.任務分配

- 在多執行緒環境中，排程需要安全地將任務分配給執行緒，避免同一任務被多個執行緒重複執行。
- CAS 可確保只有一個執行緒能成功更新共享變數（如任務索引或狀態），實現執行緒安全的任務分配。

2. 減少鎖的使用

- 傳統的鎖機制（如 Mutex）會導致執行緒進入等待狀態，增加上下文切換的開銷。
- CAS 基於無鎖操作，執行緒可以重試而不會阻塞，提高系統的並發性和效率。

3. 資源的公平分配

- 當多個執行緒需要競爭有限資源時，CAS 確保資源分配是原子的，避免競爭條件導致的錯誤。

4. 避免死鎖與活鎖

- 使用 CAS 不需要對資源進行明確的加鎖和解鎖操作，從根本上消除死鎖的可能性。
- 通過設計良好的重試或退避機制，可減少因競爭過於激烈而導致的活鎖。

3. 排程中的 CAS 應用場景

1.多執行緒任務佇列

- 在排程中，執行緒需要從佇列中安全地取出任務，或將任務添加到佇列中。
- CAS 可以保證執行緒在進行佇列操作時不會干擾其他執行緒，實現高效的無鎖佇列。

2. 共享計數器

- 計算當前正在執行的任務數量，並限制同時執行的最大任務數量。
- CAS 用於更新計數器，確保當任務完成時，資源能即時釋放供其他任務使用。

3. 任務狀態切換

- 任務在不同狀態之間切換（例如，待處理 → 處理中 → 完成）時，CAS 可用於確保狀態的正確性，避免競爭條件。

4. 使用 CAS 的優點

- 高效性：無鎖設計減少了執行緒等待和上下文切換的開銷。
- 響應性：執行緒不會因為鎖而阻塞，能更快速地完成排程操作。
- 簡化排程邏輯：避免使用複雜的加鎖和解鎖流程。
- 適用性廣泛：適用於多種場景，包括高併發系統、即時任務調度和資源分配。

5. 注意事項

- 適用場景：CAS 適合共享變數操作簡單且競爭程度不高的場景；在高競爭環境中需要結合退避機制。
- 避免活鎖：當競爭非常激烈時，執行緒可能長時間重試失敗，需設計合理的重試策略或降級機制。
- 硬體支持：CAS 操作依賴於硬體提供的原子指令，因此需要確認目標平台的支援情況。

3. 比較 semaphores 以及 condition variables 的優點、缺點、重點、使用時機

- 信號量（Semaphore）
  是一種計數信號機，用於管理對共享資源的訪問。它主要用來控制多個線程對一組共享資源的訪問權限。
  - 優點：
    - **簡單且高效**：信號量操作是原子的，能有效管理對共享資源的訪問。
    - **廣泛應用**：適用於控制並發訪問有限數量資源（例如連接池、緩衝區、隊列等）的情況。
    - **適合生產者-消費者問題**：使用信號量可以輕鬆解決生產者和消費者之間的協作問題，特別是當有多個生產者或消費者時。
  - 缺點：
    - **容易錯誤使用**：如果程序員不小心，可能會引入死鎖或競爭條件。
    - **無法精確控制條件**：信號量只管理資源的計數，無法傳遞額外的狀態信息，因此無法像條件變量那樣精細控制線程的調度。
  - 使用時機：
    - **資源管理**：當需要限制對有限資源的訪問（如緩衝區、內存池等），並且資源的使用與釋放是明確的。

        - **控制並發數量**：限制同時運行的工作線程數量（最大連接數、最大並發任務數量等）。
- 條件變量（Condition Variable）

  用於在某些條件成立時通知等待的線程，常與互斥鎖一起使用，來解決線程間的協調和同步問題。
    - 優點：
        - **精細的控制**：條件變量允許線程根據具體條件進行精細的協調，並能根據需要通知特定的線程或所有線程。
        - **簡化了複雜的同步邏輯**：當條件滿足時可以通知線程繼續工作，避免了忙等（spin-waiting）或無意義的鎖持有。
        - **靈活性高**：適合解決生產者-消費者問題，當資源不可用或需要等待某些條件時，線程能有效地等待。
    - 缺點：
        - **需要互斥鎖**：條件變量必須與互斥鎖配合使用，這樣可能會增加編程的複雜性。
        - **容易導致錯誤**：錯誤地管理互斥鎖或忘記通知等待的線程，可能會導致死鎖或線程無法被喚醒。
        - **性能開銷**：相較於信號量，條件變量的操作通常需要更多的上下文切換，特別是在條件不滿足時，會消耗一些額外的時間。
    - 使用時機：
        - **條件同步**：當某些條件需要滿足時，線程才可以繼續執行。這在多線程的協作場景中非常常見。
        - **生產者-消費者模式**：在生產者生產數據並將其放入緩衝區，消費者從緩衝區取出數據的情況下，當緩衝區空或滿時，消費者或生產者需要等待。
        - **等待事件通知**：當線程需要等待某些事件發生時，可以使用條件變量來管理這些等待和通知。
- 區別

特徵	信號量（Semaphore）	條件變量（Condition Variable）
基本概念	用來計數和控制資源訪問的數量，基於計數器。	用來在條件達成時通知等待的線程，基於事件觸發。
操作方式	直接操作計數器，阻塞或釋放線程。	線程等待或通知，與互斥鎖一起使用。
靈活性	比較簡單，適用於資源管理和限制並發數量。	更加靈活，適用於複雜的條件同步和精確控制。

通知方式	不支持條件通知，只是簡單的信號操作。	支持條件通知，可以根據狀態變化來精確控制線程。
性能開銷	操作較為簡單，不需要額外的鎖或通知機制。	需要與互斥鎖配合使用，可能增加上下文切換的開銷。
適用場景	適用於資源管理（如有限資源的訪問控制）。	適用於基於條件的等待和通知機制（如生產者-消費者問題）。
缺點	容易錯誤使用，無法精確控制條件。	需要額外的互斥鎖，容易導致死鎖或線程無法被喚醒。

第七章

1. Readers-writers problem variations 解決方法或者做法
   Reader-Writer Locks 讀者-寫者鎖
   為了解決飢餓問題，有些系統通過內核提供讀者-寫者鎖來解決這些問題。這種鎖機制可以根據不同的策略來平衡讀者和寫者的需求，確保不會出現無限等待的情況，並且可以根據具體情況決定優先權（例如，寫者優先或讀者優先）。

讀者-寫者問題（Readers-Writers Problem）

問題概述

讀者-寫者問題是一個經典的同步問題，主要關注多執行緒如何安全地訪問共享資源（例如資料庫）。該問題的核心在於如何在讀者和寫者之間進行資源的公平分配，確保數據完整性，並盡量提高並發性能。

角色定義

1. **讀者（Readers）**：
   - 讀者執行緒只需讀取共享資源，不會修改其內容。
   - 多個讀者可以同時讀取資源，不會產生衝突。
2. **寫者（Writers）**：
   - 寫者執行緒需要修改共享資源。
   - 在修改期間，必須禁止其他讀者和寫者訪問資源，以免數據不一致。

問題要求

1. 多個讀者可以同時訪問共享資源。
2. 當有寫者訪問時，必須禁止其他讀者或寫者訪問資源（寫者獨佔資源）。

3. 需要避免以下情況：
    - **讀者飢餓**：寫者不斷插隊，導致讀者長時間無法訪問資源。
    - **寫者飢餓**：讀者不斷訪問，導致寫者長時間無法修改資源。

三種變體

1. 第一類讀者-寫者問題（讀者優先）

- 當有讀者正在訪問資源時，新到的讀者可以繼續訪問，即使有寫者在等待。
- 缺點：可能導致寫者飢餓，因為讀者可能不斷進入系統。

2. 第二類讀者-寫者問題（寫者優先）

- 一旦有寫者在等待，新的讀者必須等待，優先處理寫者的請求。
- 缺點：可能導致讀者飢餓，特別是在寫者頻繁的情況下。

3. 第三類讀者-寫者問題（公平性）

- 保證請求訪問資源的順序與到達的順序一致（FIFO）。
- 這種方式能避免讀者和寫者飢餓的情況，但實現起來較為複雜。

解決方法

同步機制的使用

1. **互斥鎖（Mutex）**：
    - 控制讀者和寫者對共享資源的訪問權限，確保寫者獨佔訪問。
2. **計數器（Counters）**：
    - 使用計數器記錄當前讀者的數量，以判斷是否可以允許寫者進入。
3. **條件變數（Condition Variables）**：
    - 幫助實現讀者和寫者的等待與喚醒機制。

解決流程示例

讀者優先的簡化步驟

1. 當一個讀者想要訪問資源時：
    - 增加讀者計數器。
    - 如果有其他讀者正在讀取，允許其訪問。

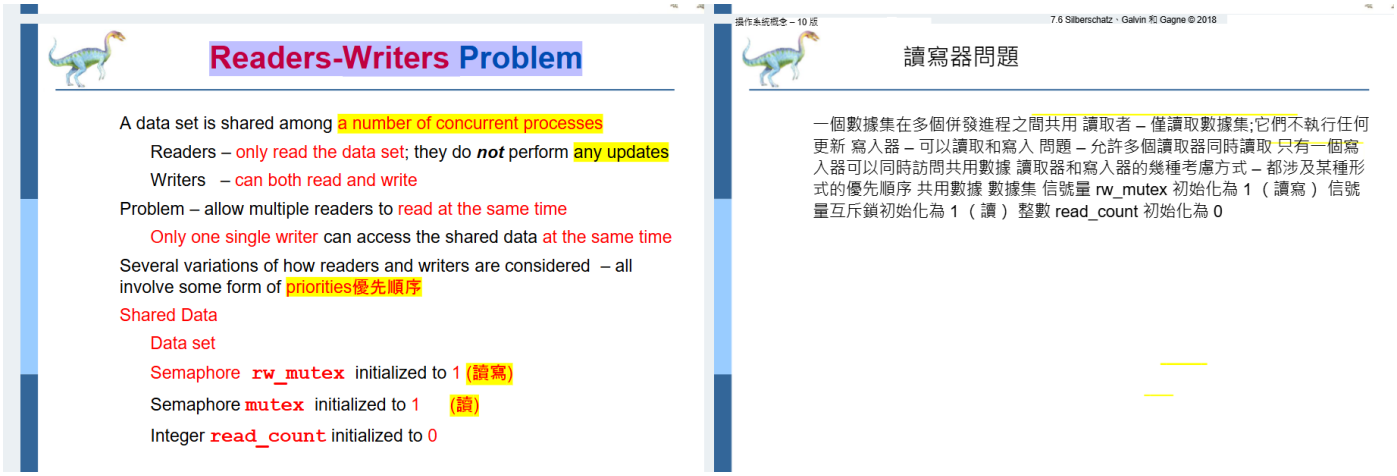
2. 當一個讀者完成訪問時：
    - 減少讀者計數器。
    - 如果讀者計數器變為 0，釋放寫者的訪問權限。
3. 當寫者想要訪問資源時：
    - 確保沒有讀者或其他寫者正在使用資源。
    - 獲取獨佔鎖後進行寫入操作。

寫者優先的簡化步驟

1. 當一個寫者想要訪問資源時：
    - 如果有其他寫者或讀者正在訪問，寫者需等待。
2. 當寫者完成訪問後：
    - 喚醒等待中的讀者或寫者，根據優先順序允許訪問。

應用場景

- **資料庫管理系統**：保證多用戶對資料庫的讀取和寫入操作互不干擾。
- **檔案系統**：允許多個應用程式同時讀取文件，但寫入時需要獨佔。
- **共享緩存**：確保同時訪問緩存的讀寫操作是安全的。



Readers-Writers Problem

A data set is shared among a number of concurrent processes

Readers – only read the data set; they do ***not*** perform any updates

Writers – can both read and write

Problem – allow multiple readers to read at the same time

Only one single writer can access the shared data at the same time

Several variations of how readers and writers are considered – all involve some form of priorities優先順序

Shared Data

Data set

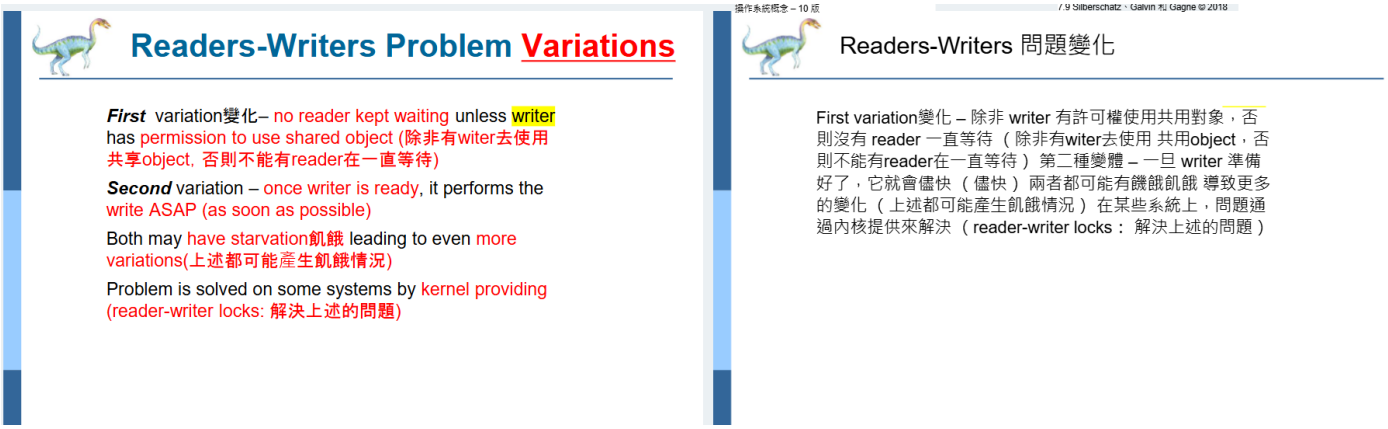
Semaphore `rw_mutex` initialized to 1 (讀寫)

Semaphore `mutex` initialized to 1 (讀)

Integer `read_count` initialized to 0

讀寫器問題

一個數據集在多個併發進程之間共用 讀取者 – 僅讀取數據集;它們不執行任何更新 寫入器 – 可以讀取和寫入 問題 – 允許多個讀取器同時讀取 只有一個寫入器可以同時訪問共用數據 讀取器和寫入器的幾種考慮方式 – 都涉及某種形式的優先順序 共用數據 數據集 信號量 rw_mutex 初始化為 1 （讀寫） 信號量互斥鎖初始化為 1 （讀） 整數 read_count 初始化為 0



2. Dining-philosophers problem 解決方法跟做法

哲學家進餐問題（Dining-Philosophers Problem）

問題概述

哲學家進餐問題是一個經典的同步問題，描述了 5 個哲學家圍坐在圓桌旁，每人左右有一支筷子。他們的行為模式是**思考**與**吃飯**交替進行，當哲學家想吃飯時，需要同時拿起左手和右手的筷子。但筷子是共享資源，如何設計方案避免死鎖與資源競爭是該問題的核心。

問題可能出現的困難

1. **死鎖**：當每個哲學家同時拿起左手筷子並等待右手筷子時，會導致系統陷入死鎖狀態。
2. **飢餓**：某些哲學家長時間無法獲得筷子，導致飢餓。
3. **資源浪費**：糟糕的同步方案可能導致哲學家等待過長時間，即使有可用筷子。

解決方法

1. 使用順序化資源分配

通過為筷子分配編號，規定哲學家只能按順序取得筷子：

- 規則：每個哲學家先拿編號小的筷子，再拿編號大的筷子。
- 特殊情況：其中一位哲學家反向操作（先拿大的，再拿小的）。

優點：

- 可以避免死鎖，因為資源取得的順序是固定的。

缺點：

- 哲學家可能長時間無法取得兩根筷子，導致飢餓。

2. 使用限制進入數量

限制同一時間內最多只有 4 個哲學家能夠拿起筷子：

- 哲學家先檢查當前有多少人在吃飯，只有當少於 4 人時才允許進入。

優點：

- 保證至少有一位哲學家能拿到兩根筷子。
- 減少資源競爭的可能性。

缺點：

- 實現稍微複雜，需要計數器來追蹤正在進行進餐的哲學家數量。

3. 使用信號量

設置一個信號量來控制筷子的使用：

- 每支筷子是一個信號量，哲學家想吃飯時需要對左右兩根筷子分別進行 **wait** 操作，吃完後執行 **signal** 操作。
- 確保信號量操作是原子的，避免多執行緒競爭筷子時發生競爭條件。

信號量算法：

- 初始化每根筷子的信號量為 1。
- 哲學家想吃飯時：
    - 對左筷子執行 **wait** 操作。
    - 對右筷子執行 **wait** 操作。
- 吃飯結束後：
    - 對左筷子執行 **signal** 操作。
    - 對右筷子執行 **signal** 操作。

優點：

- 簡單易於實現。
- 適合多執行緒的情況。

缺點：

- 可能出現死鎖，若每位哲學家同時執行 **wait** 並卡住。

4. 使用公平鎖（解決死鎖與飢餓）

採用一種公平的鎖機制，保證哲學家按照到達順序進行進餐：

- 為每位哲學家分配一個鎖，確保只有當左筷子和右筷子同時可用時，哲學家才能拿起筷子。
- 若筷子不可用，哲學家會釋放已獲得的筷子，並等待下一輪。

優點：

- 保證死鎖與飢餓不會發生。
- 確保資源分配的公平性。

5. 使用異步行為

哲學家不再必須同時拿起兩根筷子：

- 哲學家可以嘗試先拿起一根筷子，若另一根不可用則等待一段時間後重新嘗試。
- 此方法避免了每位哲學家同時拿起一根筷子的死鎖情況。

優點：

- 簡化同步邏輯。
- 減少哲學家長時間等待的機率。

比較與適用情境

解決方法	適用情境	優點	缺點
順序化資源分配	固定資源順序	簡單，避免死鎖	可能導致飢餓
限制進入數量	高併發場景	減少資源競爭，防止死鎖	實現稍微複雜
信號量	多執行緒同步	易於實現，控制資源使用	可能仍然導致死鎖
公平鎖	公平性需求高	避免死鎖與飢餓	增加等待時間
異步行為	資源競爭激烈，需求靈活的場景	減少等待時間	複雜度提升

應用場景

1. **作業系統資源分配：**
    - 如 CPU 使用權或硬碟訪問權的管理。
2. **多線程程序的資源管理：**
    - 保證多線程對共享變數或文件的訪問安全。
3. **分布式系統的鎖管理：**
    - 保證多節點對共享數據的同步訪問。

監視器（Monitor）的解法

```
monitor DiningPhilosophers {
    condition self[5];
    enum { THINKING, HUNGRY, EATING } state[5];
    procedure pickup(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) wait(self[i]);
    }
    procedure putdown(int i) {
        state[i] = THINKING;
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
    procedure test(int i) {
        if (state[(i + 4) % 5] != EATING && state[i] == HUNGRY &&
state[(i + 1) % 5] != EATING) {
            state[i] = EATING;
            signal(self[i]);
        }
    }
    procedure init() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

3. openMp 的用法重點

是一組由編譯器指令和 API（應用程式介面）所組成的工具，支持平行程式執行。

```
void update(int value)
{
    #pragma omp critical
    {
    count += value;
    }
}
```

在 #pragma omp critical 指令中的程式碼被視為臨界區，並且是原子操作（atomic），意味著它會被以不會中斷的方式執行。換句話說，當程式碼包含此指令時，該區塊的執行會保證在任何時候只有一個執行緒能進入，確保對共享變量的操作是安全的，避免了競爭條件。

OpenMP 簡介

OpenMP（Open Multi-Processing）是一個針對共享記憶體並行編程的標準，主要用於多執行緒（Multithreading）應用程式的開發。它提供了一組編譯器指令（Directives）、函數庫（Library）和環境變數，用於在 C、C++ 和 Fortran 程式中簡化並行程式的實現。

OpenMP 的特點

1. **基於共享記憶體模型**：
    - 所有執行緒共享全域記憶體，因此不需要顯式傳遞數據。
2. **簡單易用**：
    - 通過在程式中插入簡單的指令（pragma），快速將串行程式轉化為並行程式。
3. **可移植性**：
    - 支援多種硬體架構和作業系統，適合多核心處理器。
4. **動態調整執行緒數量**：
    - 執行緒數量可以根據運行時的資源動態分配。

基本概念

1. **執行緒（Thread）**：

- OpenMP 的並行性是基於執行緒，每個執行緒在程式中執行一部分工作。

2. **主執行緒（Master Thread）與從屬執行緒（Worker Threads）：**
   - 程式啟動後，主執行緒負責初始化，並根據需要產生其他執行緒參與計算。
3. **並行區域（Parallel Region）：**
   - 使用 `#pragma omp parallel` 定義程式的並行執行部分。
4. **工作分割（Work Sharing）：**
   - 通過指令將工作分配到不同執行緒。

OpenMP 的核心指令

1. **並行區域**
   - 用於指示哪部分程式需要並行執行。
   - **語法：**

```
#pragma omp parallel
{
    // 並行執行的程式區域
}
```

2. **循環並行化**
   - 自動將循環中的工作分配給多個執行緒。
   - **語法：**

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    // 每個迭代由不同執行緒執行
}
```

3. **工作分割（Sections）**
   - 將程式中的不同部分分配給不同的執行緒。

- **語法**：

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        // 第一部分工作
    }
    #pragma omp section
    {
        // 第二部分工作
    }
}
```

4. **單一執行（Single）**
    - 指定某一段程式只由一個執行緒執行。
    - **語法**：

```
#pragma omp single
{
    // 只由一個執行緒執行的程式區域
}
```

5. **臨界區（Critical）**
    - 用於保護共享資源，確保同一時間只有一個執行緒訪問。
    - **語法**：

```
#pragma omp critical
{
    // 臨界區程式
}
```

6. **Barrier（屏障）**
    - 強制所有執行緒在某點同步，等待其他執行緒完成後再繼續執行。
    - **語法**：

```
#pragma omp barrier
```

7. **Reduction（歸約操作）**
    - 將所有執行緒的結果匯總到一個變數。
    - **語法**：

```
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < n; i++) {
    sum += array[i];
}
```

OpenMP 的環境變數

1. `OMP_NUM_THREADS`：
    - 設定執行緒的數量。
    - **示例**：

```
export OMP_NUM_THREADS=4
```

2. `OMP_SCHEDULE`：
    - 控制 `for` 循環的工作分配方式（例如，靜態分配或動態分配）。

應用場景

1. **數值計算**：
    - 如矩陣運算、線性代數等。
2. **大規模數據處理**：
    - 快速排序、合併排序等可分割的算法。
3. **科學模擬**：

- 氣象模擬、分子動力學計算。

4. **圖像處理**：
    - 如濾波、邊緣檢測等。

優點

5. 易於學習和使用。
6. 高效利用多核處理器。
7. 保留串行程式的結構，便於維護。

缺點

1. 僅適用於共享記憶體模型，不適合分布式系統。
2. 某些情況下需手動調整程式以避免執行緒競爭。
3. 隨執行緒數量增加，可能出現執行緒管理的額外開銷。

第八章

1. prevention 如何改善 Deadlock

Deadlock Prevention（死鎖預防）

什麼是死鎖（Deadlock）？

死鎖是一種資源分配的問題，發生在多個進程或執行緒相互等待資源釋放的情況下，導致所有相關進程無法繼續執行。

死鎖的四個必要條件

1. **互斥（Mutual Exclusion）**：
    - 資源一次只能被一個進程使用。
2. **保持並等待（Hold and Wait）**：
    - 一個進程保持其已有資源，同時等待其他資源。
3. **不剝奪（No Preemption）**：
    - 資源不能被強制剝奪，只能由持有者主動釋放。

4. **環路等待（Circular Wait）**：
    - 存在一個進程的循環等待鏈，每個進程都等待下一個進程持有的資源。

預防死鎖的基本原則

若能破壞上述任一條件，即可避免死鎖的發生。

常用的死鎖預防方法

1. 破壞互斥條件

- **方法**：設計讓資源可以被多個進程同時共享。
- **實例**：
    - 對於只讀的資源（如文件），允許多個進程同時訪問，而不需要鎖。
    - 使用無鎖數據結構或共享變數的無鎖算法。
- **限制**：並非所有資源都能共享（如打印機、寫入文件等需要獨佔資源）。

---

2. 破壞保持並等待條件

- **方法 1**：要求進程在執行開始前請求所有需要的資源。
    - **優點**：避免了進程在持有部分資源的情況下，繼續等待其他資源。
    - **缺點**：可能會導致資源浪費，因為進程可能長時間持有未使用的資源。
- **方法 2**：若進程需要新資源，必須釋放當前持有的所有資源，然後重新請求所有需要的資源。
    - **優點**：減少進程的資源保持時間。
    - **缺點**：可能增加進程的重啟成本。

---

3. 破壞不剝奪條件

- **方法**：允許系統強制剝奪進程的資源。

- 當進程請求一個資源但無法獲取時，系統可以剝奪其已分配的資源，讓其他進程使用。

- **實例**：
  - 若某進程正在等待資源 A，而另一進程持有資源 A，但無法獲得資源 B，則可以暫時將資源 A 從該進程中剝奪，讓等待進程執行。
- **限制**：適用於能夠保存進程狀態並支持回滾的系統（如數據庫）。

---

4. 破壞環路等待條件

- **方法**：為資源分配固定的順序，進程只能按照這一順序請求資源。
  - 例如：若系統有資源 A 和資源 B，則進程只能先請求 A，再請求 B。
- **優點**：
  - 避免了循環等待鏈的產生。
- **缺點**：
  - 資源分配的靈活性降低，某些進程可能需要等候額外的時間。

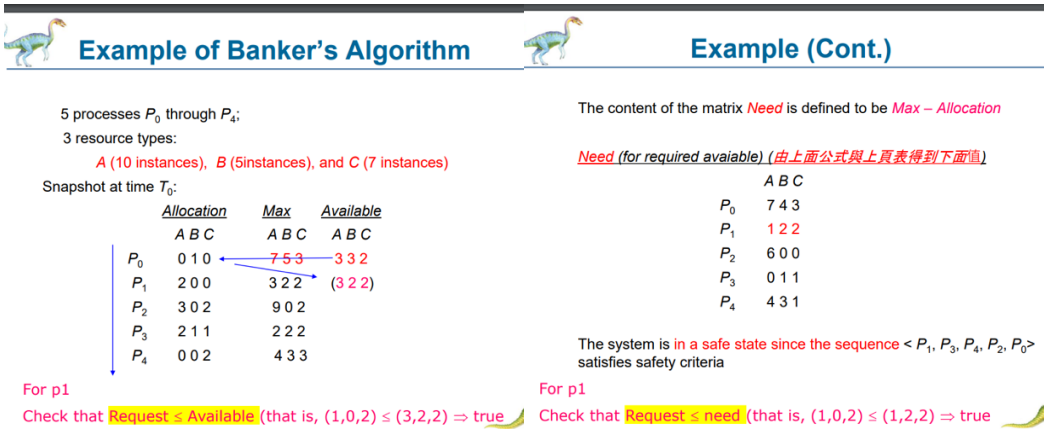
死鎖預防的實際應用

1. **操作系統資源管理**：
   - 使用資源分配圖檢測和避免死鎖。
   - 強制進程在執行前聲明所有所需資源（如 Banker's Algorithm）。
2. **數據庫管理系統**：
   - 使用鎖機制和超時策略來防止長時間等待的死鎖。
   - 實現回滾機制（例如，強制釋放資源）。
3. **多執行緒程式**：
   - 設計公平鎖和排序鎖，避免執行緒間的死鎖。
   - 優先使用無鎖的數據結構（如原子操作、Compare-and-Swap）。

總結

死鎖預防的關鍵在於從四個必要條件出發，採用合適的方法進行約束或設計，結合應用場景選擇最適合的解決方案。雖然某些方法可能會降低系統性能或靈活性，但可以顯著提高系統的穩定性與可靠性。

2. Avoidance 如何改善 Deadlock
   死鎖避免算法：銀行家算法（Banker's Algorithm）
   銀行家算法會在每次資源分配時，先檢查是否能將這些資源分配給請求進程，同時不破壞系統的安全狀態。系統會檢查所有進程的最大需求是否可以在剩餘的資源數量範圍內滿足。如果所有進程的需求都能被滿足並順利完成，那麼這次資源分配是安全的。如果任何一個進程的需求無法滿足，則分配將被拒絕，避免進入不安全狀態。
3. banker’ s Algorithm 的做法



- 進程與資源總量：系統有 5 個進程 P0 到 P4，以及三種資源類型 A,B,C，其總量分別為 10、5 和 7。
- 資源配置表：
  - Allocation：已分配給每個進程的資源量。
  - Max：每個進程最多需要的資源量。
  - Available：當前系統可用的剩餘資源量。此處為 3,3,23, 3, 23,3,2
- 檢查進程 P1：確保 Request≤Available。進程 P1 的請求是 1, 0, 2，而 Available 是 3,3,2。因此： (1,0,2)≤(3,3,2)這條件也成立。

Deadlock Avoidance（死鎖避免）

什麼是死鎖避免？

死鎖避免是一種動態資源分配策略，通過提前檢查資源分配是否可能導致死鎖，來確保系統永遠不會進入死鎖狀態。與死鎖預防不同，死鎖避免允許四個必要條件（互

斥、保持並等待、不剝奪、環路等待）部分存在，但通過系統檢測避免實際發生死鎖。

死鎖避免的核心思想

1. **系統分析**：
    - 在分配資源前，系統會模擬該資源分配是否可能導致死鎖。
    - 如果安全，則允許分配；否則，進程需要等待，直到分配不會造成死鎖。
2. **安全狀態**：
    - 系統處於安全狀態，意味著可以按照某種順序完成所有進程的資源需求，且不會發生死鎖。
3. **不安全狀態**：
    - 系統進入不安全狀態後，可能導致死鎖，但不一定立即發生。

經典的死鎖避免算法：銀行家算法（**Banker's Algorithm**）

核心概念

- 銀行家算法將系統視為銀行，進程視為客戶，資源視為貸款。
- 銀行家根據系統剩餘資源和進程最大需求，決定是否批准當前資源請求。

主要參數

1. **可用資源（Available）**：
    - 系統當前可分配的資源數量。
2. **最大需求（Max）**：
    - 每個進程在執行完成之前可能需要的資源最大數量。
3. **分配資源（Allocation）**：
    - 每個進程當前已獲得的資源數量。
4. **需求資源（Need）**：

- 每個進程尚需要的資源數量，`Need = Max - Allocation`。

算法步驟

1. **檢查資源請求是否有效**：
   - 若進程請求的資源數量超過其 `Need` 或系統 `Available` 資源數量，則拒絕請求。
2. **試探性分配**：
   - 假設資源已分配給進程，更新 `Available`、`Allocation` 和 `Need`。
3. **檢查系統是否處於安全狀態**：
   - 從剩餘資源中模擬執行每個進程，判斷是否能最終完成所有進程。
   - 若安全，完成分配；否則，回滾分配操作。

銀行家算法的示例

初始狀態

假設有：

- 3 種資源（A, B, C），初始可用數量為 `Available = [3, 3, 2]`。
- 進程需求如下表：

進程	Max (最大需求)	Allocation (已分配)	Need (需求)
P1	[7, 5, 3]	[0, 1, 0]	[7, 4, 3]
P2	[3, 2, 2]	[2, 0, 0]	[1, 2, 2]
P3	[9, 0, 2]	[3, 0, 2]	[6, 0, 0]
P4	[2, 2, 2]	[2, 1, 1]	[0, 1, 1]
P5	[4, 3, 3]	[0, 0, 2]	[4, 3, 1]

進程 **P1** 請求 **[1, 0, 2]**

1. **檢查請求是否合理**：

- `Request <= Need`：`[1, 0, 2] <= [7, 4, 3]`，成立。
- `Request <= Available`：`[1, 0, 2] <= [3, 3, 2]`，成立。

2. **試探性分配**：
   - 更新 `Available = [2, 3, 0]`。
   - 更新 `Allocation (P1) = [1, 1, 2]`。
   - 更新 `Need (P1) = [6, 4, 1]`。
3. **檢查安全狀態**：
   - 運行模擬，檢查是否存在安全序列。
   - 假設序列為 `P2 -> P4 -> P5 -> P1 -> P3`，所有進程最終都能完成，系統安全。
4. **完成分配**：
   - 請求通過，資源成功分配給 P1。

死鎖避免的其他方法

1. 資源分配圖

- 將進程和資源表示為圖結構，使用邊表示資源的請求或分配。
- 在每次資源請求時檢查圖是否出現循環。
- 若存在循環，拒絕請求。

2. 最大需求限制

- 設定每個進程的最大需求，避免進程過多佔用資源。
- 保證剩餘資源足以完成其他進程。

3. 時間限制

- 當進程請求資源時間超過一定限制時，回滾操作，釋放其已獲得的資源。

死鎖避免的優缺點

優點

1. 提高系統資源利用率。
2. 動態檢查資源分配，防止死鎖發生。
3. 適用於多進程、多資源的系統。

缺點

1. 實現複雜：需要額外的數據結構和計算來檢查安全性。
2. 資源保守：為避免死鎖，系統可能會拒絕一些合理的請求。
3. 不適用於未知資源需求的系統。

應用場景

1. **操作系統**：
    - 用於管理 CPU、記憶體和 I/O 資源。
2. **數據庫管理**：
    - 防止事務之間的鎖資源衝突。
3. **嵌入式系統**：
    - 確保有限硬體資源的有效分配。

透過死鎖避免技術（如銀行家算法），系統可以在動態資源分配過程中提前檢查，確保安全的運行環境，避免死鎖的發生，但需要在性能與實現成本之間進行權衡。

第九章

1. swapping on Mobile systems 的做法以及優缺點
    - 作法
        - iOS 的交換機制
            - 自動釋放內存：當系統內存不足時，iOS 會要求應用自動釋放記憶體。這些只讀數據會被丟棄，並且如果需要，會從閃存重新加載。

        - 終止應用：若應用無法釋放記憶體，iOS 可能會終止應用來避免進一步的內存問題。
    - Android 的交換機制
        - 狀態儲存與快速重啟：當 Android 系統內存不足時，它會先將應用的狀態寫入閃存，然後終止應用。這樣可以在系統有空間時快速重啟應用，恢復之前的狀態。
- 優點
    - **避免系統崩潰**：在內存不足時，交換可以避免系統崩潰或過慢，因為它能動態釋放不再使用的內存。
    - **提高資源利用率**：交換機制可以最大化地利用有限的內存資源，允許更多的應用同時運行。
    - **確保系統穩定性**：通過終止部分進程，系統能保持一定的穩定性，避免因為內存溢出而導致的錯誤。
- 缺點：
    - **性能開銷**：交換操作需要將數據從主記憶體移動到閃存或其他存儲設備中，這會帶來一定的性能開銷，特別是在移動設備中，閃存的讀寫速度較慢，可能影響整體性能。
    - **記憶體碎片**：長時間的交換可能會導致內存碎片化，因為交換操作需要不斷地分配和釋放內存區塊，這會使得內存的分配更加複雜。
    - **有限的寫入次數**：在移動設備中，閃存有寫入次數限制。過度的交換可能會增加閃存的寫入壽命問題，影響設備的長期使用。
    - **耗電量增加**：交換過程需要頻繁地讀寫閃存，這可能會增加設備的功耗，特別是在低效能的移動設備中。

Swapping on Mobile Systems（行動系統中的交換）

交換（Swapping）是記憶體管理的一種技術，用於在主記憶體不足時將不活躍的進程或頁面移到磁碟存儲（如交換區域）中，並在需要時再載入到記憶體。然而，在行動系統中，交換的實現和優化需要特別考慮，因為行動設備有資源限制和性能需求。

行動系統中的交換特點

1. **資源限制**
    - 行動設備通常具有較少的主記憶體和存儲空間。
    - 處理器性能相對較弱，頻繁的交換可能導致性能下降。
2. **存儲設備**

- 行動設備多使用 NAND 型快閃記憶體（如 eMMC、UFS）代替傳統硬碟。
- 快閃記憶體有寫入壽命限制，過度交換可能縮短設備壽命。

3. **實時性需求**
   - 行動應用通常需要快速響應，過多的交換可能導致應用卡頓或系統響應變慢。
4. **省電需求**
   - 頻繁的磁碟 I/O 操作會增加功耗，不利於行動設備的電池壽命。

交換的實現方式

1. 傳統交換機制

- 將整個進程移出主記憶體到交換區域，適合桌面系統。
- 在行動系統中不常使用，因為進程整體交換過於昂貴且不符合行動設備特點。

2. 按頁交換（Paging-Based Swapping）

- 只將進程的部分頁面移出主記憶體，而非整個進程。
- 更適合行動設備，因為它能夠更靈活地釋放內存。

行動系統中的交換優化

1. 壓縮交換（ZRAM）

- **概念**：
  - 使用壓縮技術將不活躍的頁面壓縮並存放在主記憶體內，而不是移到磁碟。
- **優點**：
  - 減少磁碟 I/O 操作，降低功耗。
  - 提高性能，因為壓縮與解壓操作比磁碟 I/O 快。

- **缺點**：
    - 需要額外的 CPU 資源進行壓縮和解壓縮。
- **適用場景**：
    - 行動設備廣泛使用，特別是 Android 系統。

2. 混合交換（Hybrid Swapping）

- **概念**：
    - 結合壓縮交換和傳統交換，先嘗試壓縮頁面，壓縮失敗時再將頁面移到磁碟。
- **優點**：
    - 在節省主記憶體的同時，最大化利用磁碟空間。
- **適用場景**：
    - 高記憶體需求的行動應用（如多任務處理、遊戲）。

3. 動態交換門檻

- **概念**：
    - 根據系統狀態（如記憶體佔用率、應用優先級）動態調整交換策略。
- **優點**：
    - 提升交換效率，減少對用戶體驗的影響。
- **適用場景**：
    - 任務負載多變的行動系統。

4. 寫入合併（Write Coalescing）

- **概念**：
    - 將多個小的頁面交換操作合併為一次較大的磁碟寫入操作，減少快閃記憶體磨損。
- **優點**：
    - 延長存儲設備壽命。

- 降低磁碟寫入延遲。

- **適用場景**：
  - 高頻磁碟交換場景。

行動系統交換的挑戰

1. **功耗管理**
   - 磁碟 I/O 操作會顯著增加電池消耗，需要降低交換頻率以延長設備使用時間。
2. **用戶體驗**
   - 交換可能導致應用切換卡頓，特別是在運行高記憶體需求的應用時。
3. **設備壽命**
   - NAND 型快閃記憶體有寫入壽命限制，過度交換會導致快閃記憶體過早損壞。

交換的應用場景

1. **多任務處理**
   - 當多個應用同時運行時，將不活躍的應用頁面交換到磁碟，釋放記憶體給前台應用。
2. **高記憶體需求的應用**
   - 運行大型遊戲、視頻編輯軟件等需要大量記憶體的應用時，交換有助於避免系統崩潰。
3. **低記憶體設備**
   - 在記憶體容量有限的低端行動設備上，交換技術能提升多任務能力。

行動系統交換的優缺點

優點

- **提高記憶體利用率**：允許運行更多應用或處理更大的數據集。

- **減少系統崩潰**：在記憶體不足時提供額外緩衝。
- **支持多任務處理**：釋放記憶體給前台應用，提升用戶體驗。

缺點

- **性能開銷**：交換涉及磁碟 I/O 操作，可能降低系統性能。
- **壽命影響**：快閃記憶體的寫入壽命可能因頻繁交換而縮短。
- **功耗增加**：磁碟操作會增加電池消耗，影響設備續航。

總結

行動系統中的交換機制與桌面系統相比，需要更多優化以適應資源受限的環境。壓縮交換（如 ZRAM）是行動設備中廣泛採用的技術，因為它能在降低功耗的同時提升性能。此外，動態交換門檻和混合交換策略可以根據使用情況靈活調整，進一步提升系統效率和用戶體驗。在行動設備設計中，交換技術的選擇需平衡性能、功耗和硬體壽命。

2. external Fragmentation 跟 Internal fragmentation 不一樣的地方以及發生之後的改善方法
   - 外部碎片 (External Fragmentation)
     指系統中總記憶體空間足夠來滿足某個進程的記憶體需求，但由於空閒記憶體是分散的，並且不連續，無法提供一個足夠大的連續區塊來分配給進程。
     - 改善方法
       - **內存壓縮 (Memory Compaction)**：
         將所有已分配的區塊和空閒區塊重新排列，將空閒區塊集中在一起，這樣就能夠為新的進程提供一個連續的空間。
       - **使用分頁或分段 (Paging or Segmentation)**：
         這些技術將記憶體分割成固定大小的塊（頁）或變動大小的塊（段），從而消除外部碎片的問題，因為進程可以分配不連續的內存塊。
   - 內部碎片 (Internal Fragmentation)
     指已分配的記憶體區塊大於進程實際需求的記憶體。這多餘的部分就會成為內部碎片，儘管這部分記憶體是已經分配的，但它未被進程使用。
     - 改善方法

        - **動態分配更靈活的區塊大小：**
          可以使用更靈活的記憶體分配策略（如分段分配）來減少內部碎片，讓分配的區塊大小更接近進程實際需要的大小，避免過多浪費。
        - **使用可變大小的區塊 (Variable-sized blocks)：**
          比如，動態分配記憶體區塊而不是預設的固定大小，這樣可以減少不必要的內部碎片。
- 區別：
    - **位置問題：**
        - 外部碎片是記憶體分散的問題，記憶體空間存在，但不連續。
        - 內部碎片是已分配記憶體區塊內部空間未被使用，通常是因為分配的區塊比需要的要大。
    - **處理方法：**
        - 外部碎片通常需要通過重新整理記憶體或使用分頁、分段等技術來解決。
        - 內部碎片可以通過更靈活的區塊分配來減少，或者使用動態分配策略來避免。

External Fragmentation（外部碎片）與 Internal Fragmentation（內部碎片）

兩者都是記憶體管理中常見的問題，但它們的來源與解決方式不同。

主要區別

特性	External Fragmentation（外部碎片）	Internal Fragmentation（內部碎片）
定義	由於不連續分配記憶體，導致大量小的空閒區域無法使用。	分配的記憶體塊比實際需求大，未使用的部分浪費掉。
原因	記憶體分配和釋放過程中產生碎片，空閒區域無法有效拼接。	預分配的記憶體單位過大，導致未使用部分被浪費。
發生場景	- 可變大小的記憶體分配（如分段）。 - 動態內存管理系統。	- 固定大小的記憶體分配（如分頁）。
影響	可用記憶體不足以滿足分配需求，儘管總空間足夠。	系統無法利用多餘空間，造成記憶體浪費。

可見性	較易觀察：系統中有足夠空間，但無法滿足大塊記憶體分配需求。	不易觀察：浪費的內部空間存在於分配的區塊內部。

改善方法

1. 改善 External Fragmentation

3. **壓縮（Compaction）**
    - 將記憶體中分散的空閒區域合併為連續的空間。
    - **優點**：
        - 提高可用記憶體的利用率。
    - **缺點**：
        - 開銷高，需移動大量數據，可能導致性能下降。
4. **分區分配（Partition Allocation）**
    - 使用固定大小的分區或框架來管理記憶體（如分頁）。
    - **優點**：
        - 消除了外部碎片。
    - **缺點**：
        - 可能產生內部碎片。
5. **最佳適配法（Best-Fit Allocation）**
    - 為進程分配最小的能滿足需求的空閒區域。
    - **優點**：
        - 減少碎片。
    - **缺點**：
        - 增加了查找分區的時間。
6. **次佳適配法（Next-Fit Allocation）**
    - 從上次分配的位置繼續查找適合的區域。
    - **優點**：

- 減少搜索時間。

- **缺點**：
  - 無法完全避免碎片。

---

2. 改善 Internal Fragmentation

7. **動態分配（Dynamic Allocation）**
   - 根據進程需求，分配可變大小的記憶體區塊。
   - **優點**：
     - 減少多餘的空間浪費。
   - **缺點**：
     - 可能產生外部碎片。
8. **最佳分頁大小（Optimal Page Size）**
   - 減少分頁大小，降低內部碎片，但過小的頁面可能增加頁表開銷。
   - **優點**：
     - 減少未使用的空間浪費。
   - **缺點**：
     - 需要平衡頁面大小與管理開銷。
9. **混合分配策略（Hybrid Allocation）**
   - 結合固定和動態分配策略，根據需求靈活選擇。
   - **優點**：
     - 適應性強，能兼顧內部和外部碎片問題。
10. **內存池（Memory Pool）**
    - 將記憶體分割成多個大小不同的區域池，每種池管理特定大小的區塊。
    - **優點**：
      - 減少內部碎片，適應多樣化的記憶體需求。
    - **缺點**：

- 需要額外管理開銷。

外部碎片與內部碎片的綜合處理策略

11. 結合分頁和分段

- 將記憶體分成固定大小的頁面，並為每個進程分配可變大小的分段。
- **優點**：
  - 避免了外部碎片，並且內部分段可靈活分配。
- **缺點**：
  - 實現複雜，需要頁表和段表的管理。

12. 動態內存分配算法

- 使用 Buddy System 或 Slab Allocator 等算法動態分配記憶體。
- **優點**：
  - 對內外碎片都有較好的控制。
- **缺點**：
  - 設計和管理較複雜。

比較表

特性	External Fragmentation	Internal Fragmentation
來源	記憶體分配中空閒區域分散	分配區塊內部未使用部分浪費
優化策略	壓縮、分頁、最佳適配等方法	動態分配、最佳頁面大小、內存池等方法
適用場景	動態記憶體分配，如分段系統	固定分配單位的場景，如分頁系統

結論

- **External Fragmentation** 更常出現在動態記憶體分配場景，需要壓縮或分區分配來解決。

- **Internal Fragmentation** 則是固定分配策略的問題，可通過調整分配單位大小或動態分配來優化。
- 對於現代系統，結合分頁與分段、混合分配策略通常是解決碎片問題的有效方法。

第十章

10.1Demand paging 的做法

什麼是需求分頁？

需求分頁是一種記憶體管理技術，用於虛擬記憶體系統中。它允許程式在執行時僅載入實際需要使用的頁面，而不是一次性載入整個程式的所有頁面到主記憶體。這樣可以有效節省主記憶體空間，並提高系統的資源利用率。

運作原理

1. **分頁概念**：
    - 程式被分為固定大小的頁面（Page），物理記憶體被分為相同大小的框架（Frame）。
    - 每個頁面可以載入到任何框架中。
2. **載入頁面**：
    - 當程式需要使用某一頁（Page）但該頁未在主記憶體中時，會產生**頁面錯誤（Page Fault）**。
    - 操作系統會將所需頁面從磁碟載入到主記憶體，並更新頁表（Page Table）。
3. **頁表**：
    - 每個進程都有一張頁表，用於記錄虛擬頁面和物理框架之間的映射關係。
    - 頁表會標記每個頁面是否已載入記憶體。

需求分頁的工作流程

1. **執行程式請求**：

- 程式訪問某個虛擬地址。

2. **檢查頁表**：
   - 若該虛擬地址對應的頁面已在主記憶體中，直接訪問。
   - 若該頁面不在主記憶體中（頁表標記為無效），產生頁面錯誤。
3. **處理頁面錯誤**：
   - 暫停程式執行，操作系統將所需頁面從磁碟載入到主記憶體。
   - 若主記憶體滿，則選擇一個頁面進行替換（依據頁面替換算法）。
4. **更新頁表**：
   - 將新載入的頁面標記為有效，並記錄其物理框架號。
5. **恢復執行**：
   - 當頁面載入完成後，程式繼續執行。

頁面替換算法

當主記憶體已滿時，需要替換掉一個不再使用或使用頻率較低的頁面以騰出空間。

1. **先進先出（FIFO）算法**：
   - 替換最早載入的頁面。
   - 簡單但可能導致高頁面錯誤率。
2. **最近最少使用（LRU）算法**：
   - 替換最近最少使用的頁面。
   - 更高效，但需要額外的硬體或軟體開銷來追蹤頁面使用記錄。
3. **最不常使用（LFU）算法**：
   - 替換使用次數最少的頁面。
   - 適合長時間未使用的頁面。
4. **最優算法（Optimal Algorithm）**：
   - 替換未來最長時間不被使用的頁面。
   - 理論上最佳，但無法實現（需預知未來訪問）。

需求分頁的優點

1. **高效的記憶體利用**：
    - 只載入需要的頁面，節省主記憶體空間。
2. **支援大程式**：
    - 程式大小可以超過主記憶體的容量，因為僅需載入部分頁面。
3. **快速上下文切換**：
    - 不需要在上下文切換時載入整個程式，降低開銷。
4. **靈活性**：
    - 程式執行可以動態適應不同的記憶體需求。

需求分頁的缺點

1. **頁面錯誤開銷**：
    - 頁面錯誤會導致磁碟 I/O 操作，增加延遲。
2. **頻繁的頁面替換**：
    - 若可用記憶體過少，頻繁的頁面替換會降低系統性能，稱為**抖動（Thrashing）**。
3. **額外的硬體需求**：
    - 頁表需要額外的記憶體來存儲。
    - TLB（Translation Lookaside Buffer）用於加速頁表查詢，但增加硬體複雜性。

需求分頁的實際應用

1. **作業系統**：
    - 現代作業系統（如 Windows、Linux）使用需求分頁來管理虛擬記憶體。
2. **大型應用程式**：
    - 科學計算、數據庫系統等需要處理大數據的應用。
3. **嵌入式系統**：

- 在記憶體受限的設備中，如手機或 IoT 設備，需求分頁允許更高效地利用記憶體。

與分段（Segmentation）的區別

特性	需求分頁（Demand Paging）	分段（Segmentation）
單位	固定大小的頁面	不固定大小的段
記憶體利用	更高效，因為頁面大小固定	某些場景可能導致內部碎片問題
地址空間	虛擬地址分為頁號和頁內偏移量	虛擬地址分為段號和段內偏移量
分配方式	按頁面載入	按段載入

總結

需求分頁是現代虛擬記憶體管理的核心技術。通過按需載入頁面，實現了記憶體的高效利用，同時支持執行超過物理記憶體容量的程式。然而，頻繁的頁面錯誤和替換可能降低性能，因此需要結合合適的頁面替換算法來優化系統表現。

10.2 demand paging optimization 的步驟

Demand Paging Optimizations（需求分頁優化）

需求分頁是一種有效的記憶體管理技術，但其性能可能因頻繁的頁面錯誤和替換而下降。為了減少開銷並提高系統效率，以下是一些常用的需求分頁優化策略。

---

1. 減少頁面錯誤

a. 預取頁面（Pre-Paging）

1. **概念**：

   在預測進程未來需要的頁面時，提前將這些頁面載入主記憶體，降低頁面錯誤的可能性。

2. **優化點**：

   結合程序的訪問模式（如局部性原則），預取鄰近頁面。

   使用算法分析頁面訪問的模式，如順序讀取。

b. 增加頁面大小

1. **概念**：

   增大頁面的大小，使每次載入的頁面涵蓋更多的數據。

2. **優化點**：

   對於空間局部性較強的應用（如多數科學計算），增大頁面大小可以有效降低頁面錯誤率。

3. **限制**：

   過大的頁面可能導致內部碎片增加，浪費記憶體。

---

2. 改善頁面替換策略

a. 最近最少使用（LRU, Least Recently Used）

1. **概念**：

   替換最近最少使用的頁面，因為它可能在未來也較少被使用。

2. **優化點**：

   提供接近最優替換算法的性能。

3. **實現方法**：

   使用硬體支持的計數器或軟體模擬。

b. 頻率替換（LFU, Least Frequently Used）

1. **概念**：

   替換使用頻率最少的頁面。

2. **優化點**：

   適用於需要長時間保留高頻使用頁面的場景。

3. **限制**：

   可能導致較新的頁面被頻繁替換。

c. 混合策略

1. **概念**：

   結合多種替換算法（如 LRU + LFU），兼顧短期和長期的頁面使用情況。

---

3. 提高磁碟 I/O 效率

a. 使用 SSD 或更快的儲存設備

1. **概念**：

   將頁面存儲在速度更快的固態硬碟（SSD）或記憶體型存儲設備（如 NVMe）。

2. **優化點**：

   減少頁面載入的延遲時間。

   對 I/O 密集型應用尤其有效。

b. 磁碟讀取優化

1. **概念**：

   使用批量 I/O 操作一次載入多個頁面。

2. **優化點**：

   降低單次頁面讀取的固定開銷。

   使用非同步 I/O 技術提高並行性。

---

4. 增強頁面局部性

a. 編譯器優化

1. **概念**：

   通過優化程式的數據結構和訪問模式，使其更符合局部性原則（如空間和時間局部性）。

2. **優化點**：

使用連續的數據結構（如陣列）而非分散的鏈結串列。

提高循環的數據訪問模式的連續性。

b. 緩存策略

1. **概念**：

在主記憶體之上增加緩存層（如 CPU Cache），減少主記憶體的頁面訪問次數。

2. **優化點**：

加快頁面訪問速度。

避免頻繁的頁面替換。

---

5. 減少抖動（Thrashing）

a. 增加主記憶體

1. **概念**：

提高主記憶體容量，減少頁面替換的頻率。

2. **優化點**：

對於多任務系統尤為有效，減少執行緒之間的競爭。

b. 負載控制

1. **概念**：

限制系統中同時執行的進程數量，減少對主記憶體的過度使用。

2. **優化點**：

當記憶體負載過高時，暫停部分進程的執行。

3. **實現方法**：

使用分頁工作集模型（Working Set Model）來追蹤每個進程的活躍頁面數量。

6. 提高硬體支持

a. Translation Lookaside Buffer (TLB)

1. **概念**：

   使用硬體緩存來加速頁表查詢。

2. **優化點**：

   減少頻繁訪問頁表的開銷。

3. **限制**：

   TLB 空間有限，無法存儲所有頁面映射。

b. 多層頁表

1. **概念**：

   使用多層頁表結構來分解大型頁表。

2. **優化點**：

   減少單層頁表帶來的記憶體開銷。

   提升頁表查詢效率。

---

7. 預防和快速恢復頁面錯誤

a. 預測模型

1. **概念**：

   使用機器學習模型預測程序未來的頁面訪問模式，提前載入。

2. **優化點**：

   減少頁面錯誤的發生。

b. 快速恢復

1. **概念**：

優化頁面錯誤處理過程，減少載入頁面的延遲。

2. **實現方法**：

減少頁面錯誤處理所需的系統呼叫次數。

需求分頁優化的整體效果

優化策略	主要優勢	適用場景
減少頁面錯誤	提升系統穩定性，減少延遲	程式訪問模式明確的系統
改善頁面替換策略	提高資源利用率，減少不必要替換	多任務環境，高併發系統
提高磁碟 I/O 效率	降低頁面載入延遲	I/O 密集型應用
增強頁面局部性	提升數據訪問效率，減少頁面切換	記憶體訪問密集的應用
減少抖動	保證系統穩定性，避免性能下降	記憶體負載高的多任務系統

結論

通過結合頁面預取、替換策略優化、局部性提升和硬體支持，需求分頁的性能可以得到顯著提升。同時，針對具體應用場景選擇適合的優化技術，是實現高效記憶體管理的關鍵。

10.3 copy on write 的做法跟優缺點

Copy-On-Write (COW)

Copy-On-Write（寫時複製）是一種資源管理技術，應用於記憶體和文件系統中，目的是在多個進程或執行緒共享相同資源的情況下，延遲對資源的實際複製，直到需要進行修改時才進行真正的複製操作。這樣可以大幅降低資源的使用量和複製的開銷。

Copy-On-Write 的做法

1. **初始狀態**：
   多個進程或執行緒共享同一資源，資源設置為**只讀**，指向相同數據。
2. **資源修改請求**：
   寫操作觸發**保護錯誤**，系統執行以下操作：

   - 分配資源的副本。
   - 修改進程使用副本，原資源繼續共享。

3. **資源更新**：
   只有在寫操作時，才進行實際複製和修改操作，確保共享過程不受影響。

Copy-On-Write 的應用場景

1. **記憶體管理**
   - **操作系統中的分頁**：
     - 父進程和子進程共享記憶體頁面，直到其中一方需要修改頁面時才複製。
     - 用於 `fork()` 系統呼叫的實現。
   - **虛擬機和容器**：
     - 多個虛擬機或容器共享基礎的只讀映像，只有在需要修改時才創建副本。
2. 文件系統
   - **快照技術**：
     - 文件系統（如 ZFS、Btrfs）中的快照功能，允許用戶快速創建數據的只讀副本，只有在進行寫操作時才會複製數據塊。
   - **分散式文件系統**：
     - 在分散式文件系統中實現節約存儲的數據共享機制。
3. **資料結構**
   - **函數式編程語言**：
     - 用於實現不可變數據結構，僅在修改時創建副本。

Copy-On-Write 的優缺點

優點

1. **資源節約**
    - 多個進程共享相同的數據，減少了記憶體和存儲的浪費。
2. **提升性能**
    - 延遲了資源的複製操作，避免不必要的數據拷貝。
    - 減少初始資源分配的時間，特別是在創建子進程時。
3. **簡化數據管理**
    - 用戶無需擔心數據共享的細節，系統自動處理複製和分配。
4. **支持高效快照**
    - 文件系統中的快照技術可以快速保存數據的狀態，而不需要立即複製所有數據。

缺點

1. **寫操作的額外開銷**
    - 當進程進行寫操作時，會引發頁面錯誤並觸發數據的複製，增加了延遲。
2. **資源競爭**
    - 多個進程可能同時修改共享資源，導致頻繁的資源複製和內存管理操作。
3. **實現複雜度**
    - 系統需要追蹤哪些資源是共享的，哪些需要被複製，增加了實現的難度。
4. **存儲碎片問題**
    - 在文件系統中，寫時複製可能導致數據塊分散，增加存儲的碎片化程度。

5. **適用場景有限**
    - 僅適用於資源修改頻率較低的情況；如果頻繁修改，COW 的優勢會減弱。

Copy-On-Write 的改進措施

1. **結合預取策略**
    - 在預測即將發生的寫操作時，提前複製數據，減少頁面錯誤的頻率。
2. **減少碎片化**
    - 文件系統中，使用優化的分配算法來降低數據塊碎片化的影響。
3. **高效的頁面追蹤**
    - 使用更快的數據結構（如哈希表或位圖）來管理共享頁面，提高記憶體管理效率。
4. **靈活的複製策略**
    - 在多核系統中，分配不同的 CPU 和記憶體資源來處理 COW 操作，減少競爭。

總結

Copy-On-Write（COW） 是一種有效的延遲複製技術，適合多數需要共享數據的場景，特別是在修改操作相對較少的情況下。它在提升系統資源利用率和性能的同時，也面臨寫操作延遲和實現複雜度的挑戰。合理的場景選擇和策略改進是充分發揮 COW 優勢的關鍵。

10.4 page replacement 的使用時機跟優缺點

Basic Page Replacement（基本頁面替換）

頁面替換是需求分頁中處理主記憶體不足的重要機制。在進程執行時，如果所需頁面不在主記憶體中（即發生頁面錯誤），且記憶體已滿，則需要替換掉一個現有的頁面以騰出空間載入新頁面。不同算法有其適合的使用場景和優缺點。

頁面替換的流程

1. **頁面錯誤發生**：

- 當進程訪問的頁面不在主記憶體中，發生頁面錯誤。

2. **檢查記憶體狀態**：
   - 如果主記憶體中有空閒框架，直接載入所需頁面。
   - 如果記憶體已滿，則需選擇一個頁面進行替換。
3. **頁面替換算法執行**：
   - 根據算法選擇一個頁面進行替換。
   - 若被替換的頁面已被修改，需先將其寫回磁碟。
4. **載入新頁面**：
   - 將所需頁面從磁碟載入到被替換的框架中，更新頁表。
5. **恢復執行**：
   - 進程繼續執行，訪問新載入的頁面。

頁面替換演算法

1. 先進先出（FIFO, First-In-First-Out）

- **描述**：
  - 按照頁面載入記憶體的順序進行替換，最早進入記憶體的頁面最先被替換。
- **特點**：
  - 簡單易實現，只需維護一個 FIFO 隊列。
  - 可能出現 **Belady's Anomaly**，即增加頁框數反而導致更多頁面錯誤。
- **使用時機**：
  - 小型系統或對性能要求不高的場景。

---

2. 最優演算法（Optimal Algorithm）

- **描述**：

- 替換未來最長時間不會被使用的頁面，理論上能達到最少頁面錯誤。

- **特點**：
  - 理論最佳，但需要預測未來的頁面訪問模式，無法實際實現。
- **使用時機**：
  - 作為其他算法的性能比較基準。

---

3. 最近最少使用（LRU, Least Recently Used）

- **描述**：
  - 替換最近最少被訪問的頁面，假設這些頁面未來使用的可能性較低。
- **特點**：
  - 基於時間局部性，性能通常優於 FIFO。
  - 需要額外硬體支持（如計數器或堆疊）以追蹤頁面訪問時間，增加實現成本。
- **使用時機**：
  - 記憶體訪問模式穩定，且系統資源足夠支持的情況。

---

4. LRU 近似演算法（LRU Approximation Algorithms）

- **描述**：
  - 利用參考位（Reference Bit）和修改位（Modify Bit）實現簡化版本的 LRU。
  - 具體包括：
    - **Second-Chance Algorithm**（時鐘替換演算法）：基於 FIFO，檢查參考位，若參考位為 0，直接替換，若為 1，清除參考位並跳過。
    - **Enhanced Second-Chance Algorithm**：考慮參考位和修改位的組合，分為四個類別（如未被修改且未被引用的頁面優先替換）。

- **使用時機**：
    - 硬體資源有限時，提供 LRU 的簡化替代方案。

---

5. 最少使用頻率（LFU, Least Frequently Used）

- **描述**：
    - 替換使用次數最少的頁面。
- **特點**：
    - 適合長期使用模式，但無法處理臨時高頻使用頁面的情況。
- **使用時機**：
    - 需要頻繁統計頁面使用頻率的系統。

---

6. 頻繁使用演算法（MFU, Most Frequently Used）

- **描述**：
    - 替換使用次數最多的頁面，基於這些頁面可能已經完成主要用途。
- **特點**：
    - 與 LFU 相反，適用於特定訪問模式的場景。

---

7. 頁面緩衝演算法（Page-Buffering Algorithm）

- **描述**：
    - 維護一個空閒框架池，當需要替換頁面時，先將頁面加入池中，之後再進行正式替換。
- **特點**：
    - 減少頁面錯誤處理的延遲，降低錯誤選擇的影響。
- **使用時機**：
    - 高併發場景或需要快速響應的系統。

8. 分配策略（Allocation Policies）

- **全域替換（Global Replacement）**：
  - 進程可以從所有可用頁框中選擇替換目標。
  - 提高系統整體吞吐量，但會導致進程性能波動。
- **本地替換（Local Replacement）**：
  - 進程只能替換其自身分配的頁框。
  - 保證進程性能穩定，但可能導致記憶體利用率下降。

頁面替換演算法比較

演算法	優點	缺點	適用場景
FIFO（First-In-First-Out）	- 簡單易實現。 - 只需維護一個 FIFO 隊列。	- 容易出現 Belady's Anomaly（增加頁框數反而導致更多頁面錯誤）。 - 不考慮頁面使用情況。	- 小型系統或對性能要求不高的場景，例如嵌入式系統或學術研究中的測試系統。
Optimal Algorithm（OPT）	- 理論上頁面錯誤最少，性能最佳。	- 無法實現，因為需預測未來的頁面訪問模式。	- 作為其他算法的性能比較基準或理論分析的模擬環境。
LRU（Least Recently Used）	- 遵循時間局部性，通常性能優於 FIFO。 - 減少不常用頁面的替換次數。	- 實現複雜，需要額外硬體（如計數器或堆疊）來追蹤頁面訪問時間。	- 記憶體訪問模式穩定且硬體資源足夠的場景，例如高性能計算和數據庫系統。

LRU Approximation（近似 LRU）	- 提供 LRU 的簡化實現，如 Second-Chance Algorithm（時鐘替換算法）。	- 不如 LRU 精準，可能在某些場景下性能下降。	- 資源有限或無法支持完整 LRU 實現的系統，例如嵌入式設備或簡化版的多任務系統。
LFU（Least Frequently Used）	- 適合長期穩定訪問模式的場景，對低頻訪問頁面替換有優勢。	- 不能處理臨時高頻使用頁面，可能導致次數累積錯誤。	- 須頻繁統計訪問次數的系統，例如數據分析和定期運算的工作流。
MFU（Most Frequently Used）	- 假設高頻訪問頁面已完成主要用途，適合特定訪問模式。	- 適用場景有限，對普通訪問模式可能表現不佳。	- 特定訪問模式的系統，例如短期任務執行後需要釋放記憶體的情境。
Page-Buffering Algorithm	- 減少頁面錯誤處理延遲。 - 提高替換效率並降低錯誤選擇的影響。	- 增加記憶體開銷來維護緩衝池，實現相對複雜。	- 高併發場景或需要快速響應的系統，例如多任務操作系統或大型線上服務。
Global Replacement	- 提高系統整體吞吐量。	- 可能導致進程性能波動，進程之間相互影響較大。	- 需要優化整體性能而非單個進程的場景，例如多用戶共享資源的系統。
Local Replacement	- 保證進程性能穩定，避免進程之間相互干擾。	- 整體記憶體利用率可能下降，進程之間的空間分配不靈活。	- 需要穩定性能的場景，例如關鍵任務進程或實時系統。

圖形化比較（建議用圖表呈現）

- **X 軸**：算法的複雜度（低到高）
- **Y 軸**：性能（頁面錯誤率，低到高）

算法	性能表現	複雜度
FIFO	一般	低

Optimal (OPT)	最佳（理論基準）	高（不可實現）
LRU	優秀	高
LRU Approximation	良好	中
LFU	穩定	中
MFU	特定場景中穩定	中
Page-Buffering	優秀	高

結論

1. **簡單場景**：
    - 使用 **FIFO** 或簡化的 **LRU Approximation**，適合硬體資源有限或性能要求不高的系統。
2. **高性能需求**：
    - 優先選擇 **LRU** 或 **Page-Buffering Algorithm**，特別適合高性能計算或多任務併發場景。
3. **理論分析**：
    - 使用 **Optimal Algorithm (OPT)** 作為基準，幫助衡量其他算法性能。
4. **特殊場景**：
    - **LFU** 和 **MFU** 適用於需要基於訪問頻率調整的系統。

性能優化建議

1. **結合訪問模式**：
    - 根據程序的時間或空間局部性特徵，選擇適合的替換算法。
2. **增加頁框數量**：
    - 為頻繁訪問的進程分配更多頁框，減少頁面錯誤。
3. **使用預取（Prefetching）**：
    - 預測程序可能訪問的頁面，提前載入，降低頁面錯誤率。
4. **多級緩存結構**：

- 利用 CPU 緩存或 TLB（Translation Lookaside Buffer）來加速頁表查詢，減少頁面替換的影響。

Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
   Set validation bit = **v**
5. Restart the instruction that caused the page fault