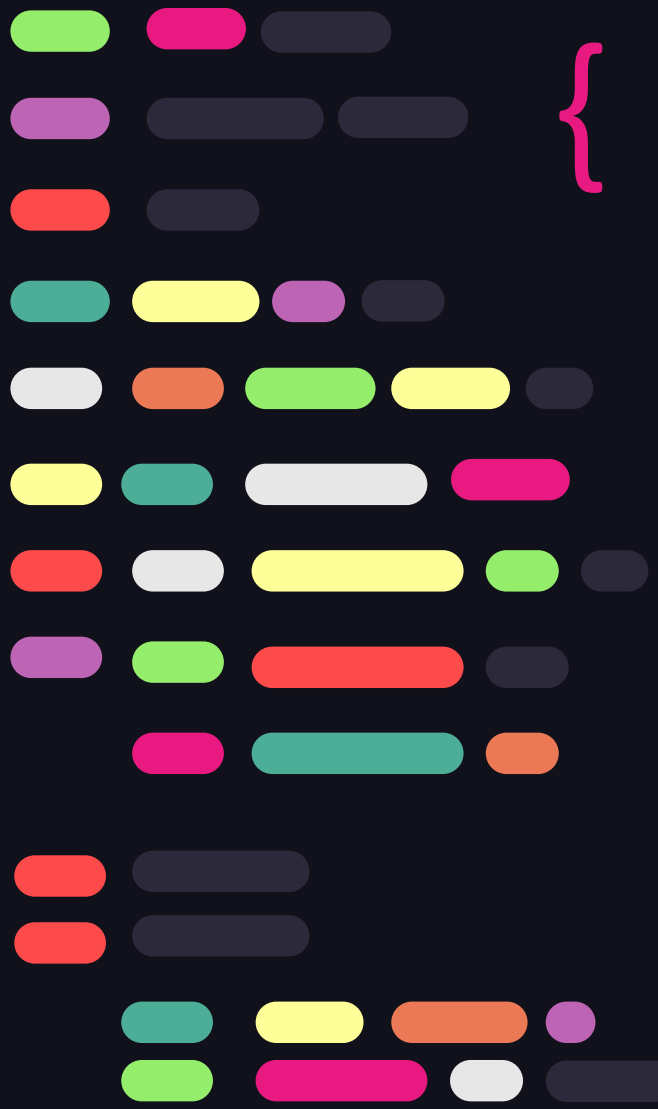
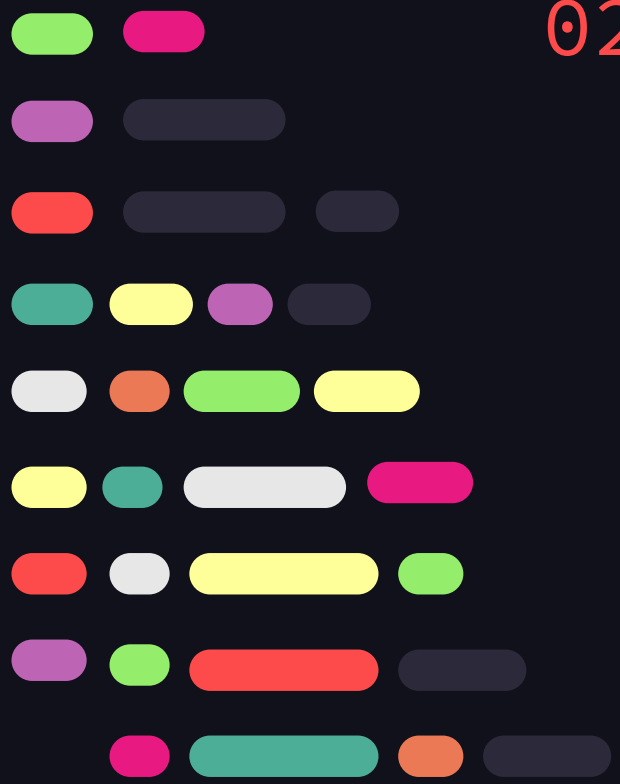
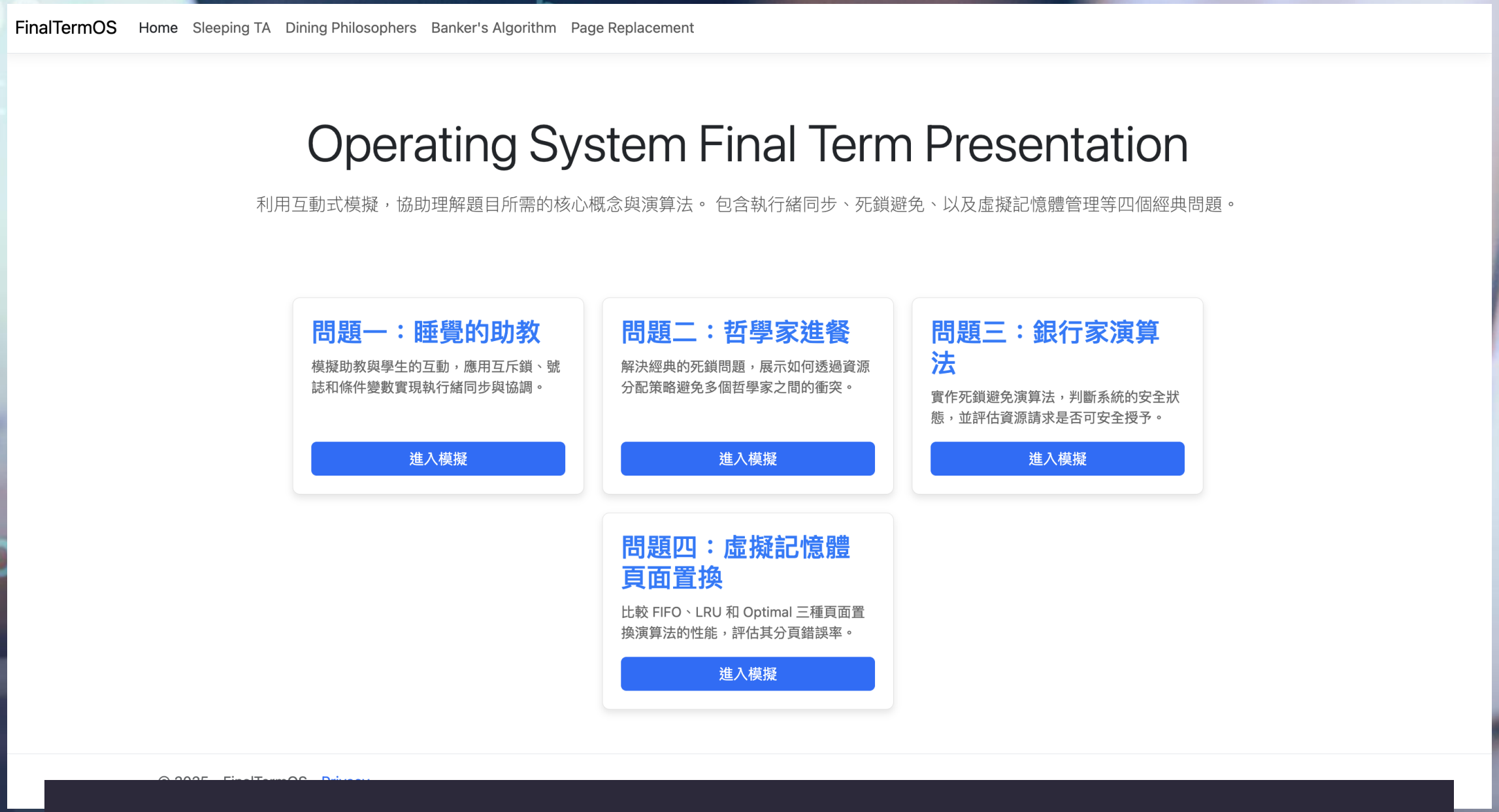


Operating System Final Project Presentation

<9B117023 張庭睿>

目錄

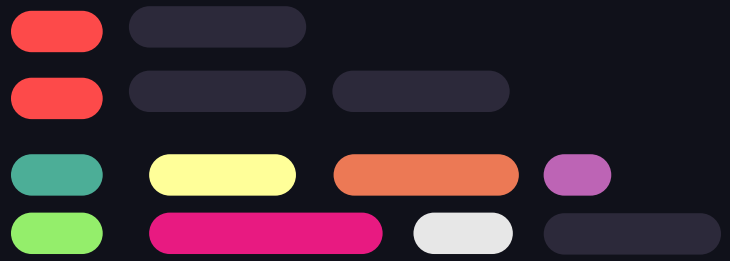



FinalTermOS
Home
Sleeping TA
Dining Philosophers
Banker's Algorithm
Page Replacement
Operating System Final Term Presentation
利用互動式模擬，協助理解題目所需的核心概念與演算法。 包含執行緒同步、死鎖避免、以及虛擬記憶體管理等四個經典問題。
問題一：睡覺的助教
模擬助教與學生的互動，應用互斥鎖、號誌和條件變數實現執行緒同步與協調。
進入模擬
問題二：哲學家進餐
解決經典的死鎖問題，展示如何透過資源分配策略避免多個哲學家之間的衝突。
進入模擬
問題三：銀行家演算法
實作死鎖避免演算法，判斷系統的安全狀態，並評估資源請求是否可安全授予。
進入模擬
問題四：虛擬記憶體頁面置換
比較 FIFO、LRU 和 Optimal 三種頁面置換演算法的性能，評估其分頁錯誤率。
進入模擬

Home Page

Sleeping TA 問題陳述

一名助教 (TA) 在辦公室內服務學生。
走廊有 6 把椅子供等待的學生使用。
TA 在無學生時小睡；學生到達發現 TA 睡覺時需叫醒 TA。
學生到達發現 TA 正忙時，坐椅子等待；無椅子則稍後回來。
TA 服務完畢後，必須檢查隊列是否有學生等待，並輪流服務；無學生則回小睡。



•執行緒 (Threads)：
- TA 和每位學生各自運行為獨立的執行緒。

•目的：
- 模擬多個並行活動單元，各自執行其邏輯。
(如:思考、服務、等待)

•共用資源：
- 走廊的椅子數量 (有限資源)。
- 等待學生的隊列 (_waitingStudentsQueue)。
- TA 的服務時間。

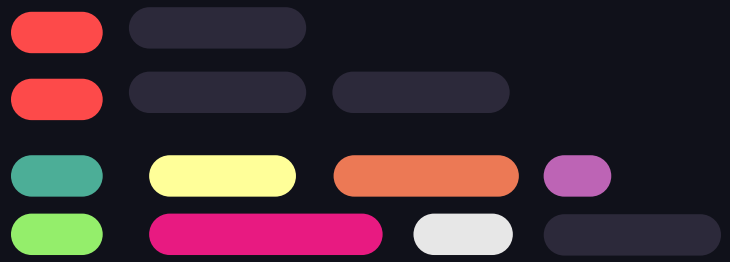
Key Concept

 # 互斥鎖 (Mutex / lock)：lock (_lock)
•目的：
- 確保**共用數據**（如等待隊列 、TA 和學生的**狀態變數**
在任何時刻只被一個執行緒存取。

•機制：
- 防止多個執行緒同時修改數據導致競爭條件 (Race Condition) 和數據不一致。

 # 號誌 (Semaphore / SemaphoreSlim)：
•_waitingChairsSemaphore （計數型號誌）：
- **•目的：** 管理走廊上有限的椅子數量。
- **•機制：** 學生 Wait() 嘗試佔用椅子，TA Release() 釋放椅子。

•_taSemaphore （二進制型號誌，作為事件信號）：
- **•目的：** 讓學生能「叫醒」正在睡覺的 TA。
- **•機制：** TA Wait() 進入睡覺狀態，學生 Release() 喚醒 TA。



 # 條件變數 (Conditional Variable)：
=> student.WaitingSignal
•目的：
- 讓學生執行緒在成功坐上椅子後，
精確地等待被 TA 服務，直到輪到自己時才被喚醒。

•機制：
- 學生入隊後 student.WaitingSignal.Wait() 阻塞；
TA 服務學生時 student.WaitingSignal.Set()
喚醒特定學生。

畫面展示

可以調整學生總數以及走廊椅子的數量，並可切換學生被服務完之後是否繼續循環的開關。

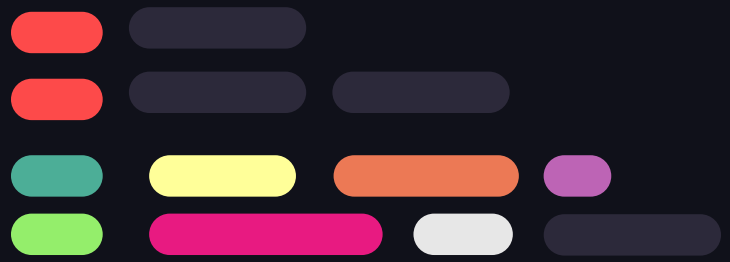
在模擬狀態下，可以看得到TA的狀態以及椅子被占用狀況，且等待對列的人數也會顯示在下方；學生的狀態顏色會因為不同的動作而改變，若循環開關關閉則會有「被幫助，已離開」顯示。

Dining Philosophers 問題陳述

- 9 位哲學家圍坐圓桌，每人左右各有一枝筷子。
- 哲學家交替**思考**與**吃飯**；吃飯需要同時擁有左右兩枝筷子。

- 每次只能拿取一枝筷子。

- 如果無法同時拿起兩枝筷子，
  則必須**放下**已拿到的筷子，
  然後重新思考並稍後再試。



 # 執行緒 (Threads)：
- 每位哲學家各自運行為獨立的執行緒。
- **目的：** 模擬多個並行活動單元，交替執行「思考」和「吃飯」的生命週期。

 # 共用資源：
- 桌上的筷子（數量與哲學家數量相同）。每根筷子都是一個必須獨佔的資源。

Key Concept

 # 經典問題：死鎖 (Deadlock) 風險：
- 如果所有哲學家都同時拿起左邊的筷子，並等待右邊的筷子，則會發生循環等待，導致**死鎖**。
- **條件：**
  互斥 (Mutal exclusion)、持有並等待 (Hold and wait)、不可搶佔 (No preemption)、循環等待 (Circular wait).

 # 死鎖避免策略 (Deadlock Avoidance Strategy)：
- **策略核心：「不滿足條件則放棄已獲資源」。**
- **機制：** 哲學家會先嘗試拿起左邊的筷子。
  如果成功拿到左邊，再嘗試拿右邊。
  如果右邊的筷子不可用，
  則哲學家必須立即放下已拿到的左邊筷子，
  然後進入等待（隨機延遲）狀態，稍後再重新嘗試。
  這確保了不會有哲學家長期持有資源並等待，
  打破了「持有並等待」條件。

 # 程序上的應用：
- **互斥鎖 (Mutex)：(lock)**
  - **目的：** 保護**筷子的狀態**，
    確保在任何執行緒檢查或修改筷子可用性時，
    都能獨佔存取，避免**競爭條件**。
  - **機制：** 哲學家在嘗試拿取左右兩側筷子時，
    都必須在 lock的保護下進行判斷和修改筷子的
    IsAvailable 狀態，確保操作的原則性。

 # 條件變數 (ManualResetEventSlim)：
=> philosopher.CanEatSignal
在本方案中，雖然哲學家在無法拿到第二枝筷子時，
是通過**放下已拿到的筷子**並 Task.Delay
一段時間來實現等待和重試，而不是直接在此變數上阻塞。

畫面展示

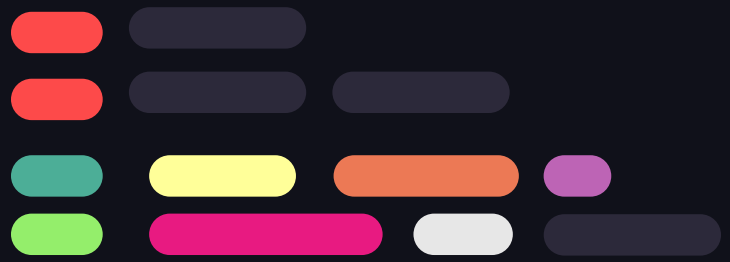
可以調整哲學家的數量，筷子數量也會跟著變更，並可在模擬過程中暫停模擬，觀看個別哲學家狀態。

在模擬狀態下，可以看得到哲學家有無拿起筷子及進食，和筷子的是否被拿取。

下方的模擬日誌，能夠實時地看得到哲學家的動作。

Banker's Algorithm 問題陳述

- 實作銀行家演算法，
  這是一種避免死鎖發生的**排程演算法**。
- 透過檢查系統是否處於**安全狀態**，
  來判斷資源請求是否可立即滿足。
- 給定特定資源類型 (A, B, C, D) 和
  程序 (P1-P5) 的初始可用、最大需求和已分配資源量。



 # 死鎖避免 (Deadlock Avoidance)：
- 演算法在資源分配前進行預檢查，
  確保分配不會導致不安全狀態，從而避免死鎖。

 # 安全狀態 (Safe State)：
- 系統處於安全狀態，如果存在一個安全序列 (Safe Sequence)，
  使得所有進程都能按序完成執行並釋放資源，而不會導致死鎖。

 # 核心矩陣/向量：
- Available （可用資源向量）：當前系統中每種資源的可用實例數量。
- Max （最大需求矩陣）：每個進程對每種資源的**最大需求量**。
- Allocation （分配矩陣）：目前已分配給每個進程的每種資源數量。
- Need （需求矩陣）：每個進程還需要的每種資源數量，
  計算方式為 Need = Max - Allocation。

Key Concept

 # 安全演算法 (Safety Algorithm)：

- 目的：判斷系統是否處於安全狀態，
並找出所有可能存在的安全序列。

- 機制：迭代檢查尚未完成的進程，
若某進程的 Need 小於等於當前 Work （可用資源），
則假設其完成並釋放其已分配資源，更新 Work，
遞歸尋找所有可完成的進程順序。

 # 資源請演算法 (Resource-Request Algorithm)：

- 目的：當進**求**程 P{n} 請求資源時，
判斷系統是否能**立即安全地滿足**請求。

- 機制：

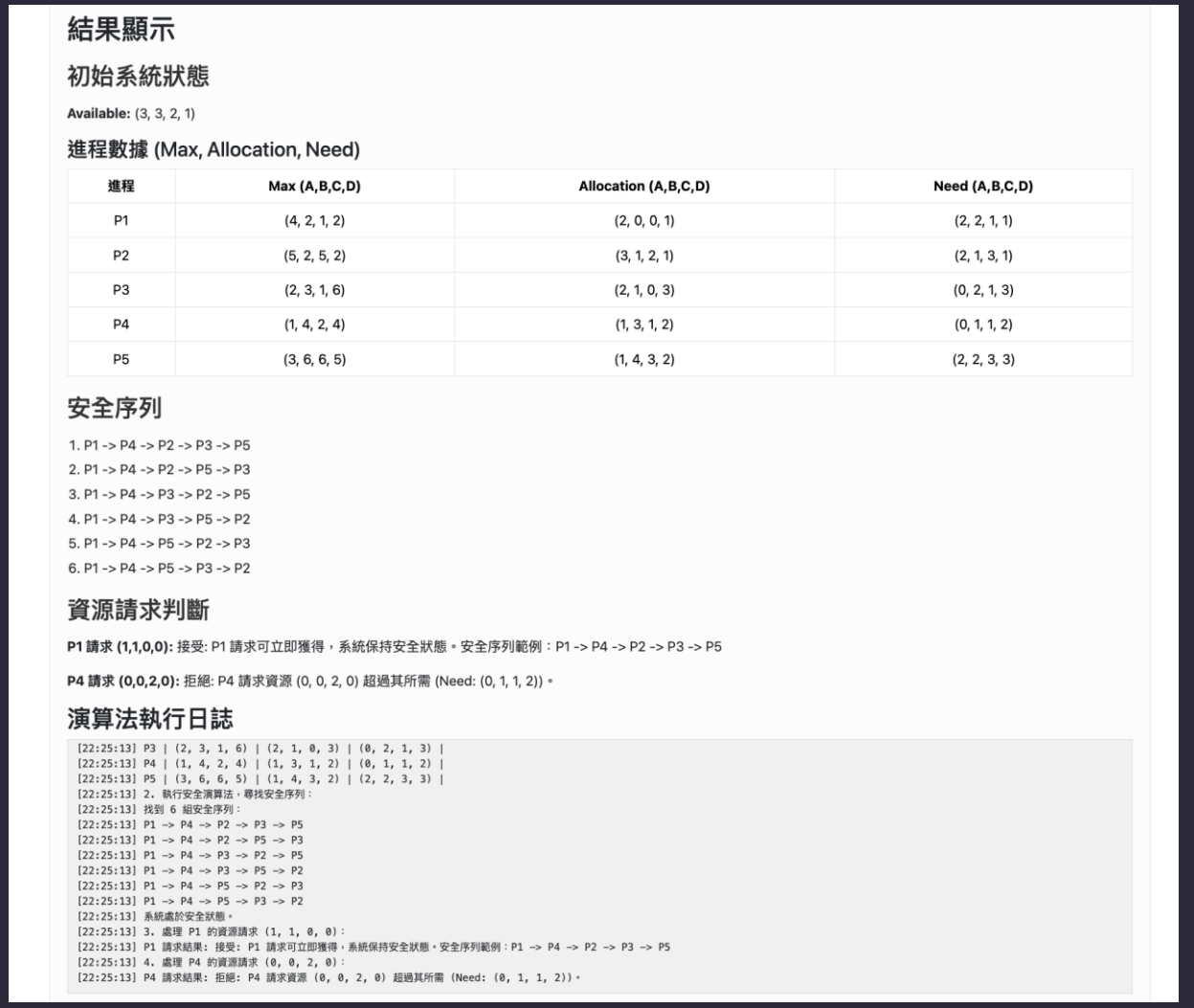
1. 檢查請求量是否合法
   (Request <= Need)
2. 檢查請求量是否在當前可用範圍內
   (Request <= Available)
3. 若兩者皆滿足，**假設**資源已分配
   (更新 Available、Allocation、Need)
   => 然後運行**安全演算法**。
4. 若安全演算法判斷系統仍安全，則接受請求；
   否則，拒絕請求，進程必須等待。

畫面展示

由於題目有正確固定答案，在系統初始狀態下可以看得到題目內容。

且可以透過安全序列得知由安全演算法得出的序列排列可能。

並顯示資源請求判斷的演算法執行日誌，可以看得到兩種演算法的推導及過程。



結果顯示

初始系統狀態

Available: (3, 3, 2, 1)

進程數據 (Max, Allocation, Need)

進程	Max (A,B,C,D)	Allocation (A,B,C,D)	Need (A,B,C,D)
P1	(4, 2, 1, 2)	(2, 0, 0, 1)	(2, 2, 1, 1)
P2	(5, 2, 5, 2)	(3, 1, 2, 1)	(2, 1, 3, 1)
P3	(2, 3, 1, 6)	(2, 1, 0, 3)	(0, 2, 1, 3)
P4	(1, 4, 2, 4)	(1, 3, 1, 2)	(0, 1, 1, 2)
P5	(3, 6, 6, 5)	(1, 4, 3, 2)	(2, 2, 3, 3)

安全序列

1. P1 -> P4 -> P2 -> P3 -> P5
2. P1 -> P4 -> P2 -> P5 -> P3
3. P1 -> P4 -> P3 -> P2 -> P5
4. P1 -> P4 -> P3 -> P5 -> P2
5. P1 -> P4 -> P5 -> P2 -> P3
6. P1 -> P4 -> P5 -> P3 -> P2

資源請求判斷

P1 請求 (1,1,0,0): 接受: P1 請求可立即獲得，系統保持安全狀態。安全序列範例：P1 -> P4 -> P2 -> P3 -> P5

P4 請求 (0,0,2,0): 拒絕: P4 請求資源 (0, 0, 2, 0) 超過其所需 (Need: (0, 1, 1, 2))。

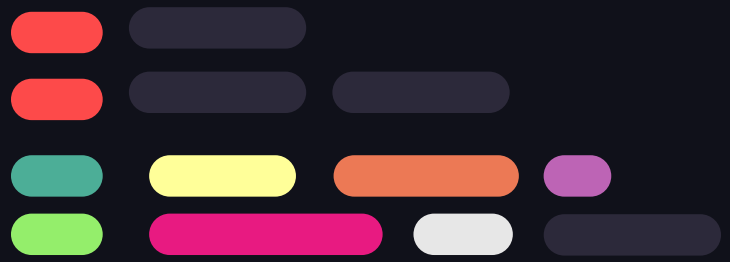
演算法執行日誌

```
[22:25:13] P3 | (2, 3, 1, 6) | (2, 1, 0, 3) | (0, 2, 1, 3) |
[22:25:13] P4 | (1, 4, 2, 4) | (1, 3, 1, 2) | (0, 1, 1, 2) |
[22:25:13] P5 | (3, 6, 6, 5) | (1, 4, 3, 2) | (2, 2, 3, 3) |
[22:25:13] 2. 執行安全演算法，尋找安全序列：
[22:25:13] 找到 6 組安全序列：
[22:25:13] P1 -> P4 -> P2 -> P3 -> P5
[22:25:13] P1 -> P4 -> P2 -> P5 -> P3
[22:25:13] P1 -> P4 -> P3 -> P2 -> P5
[22:25:13] P1 -> P4 -> P3 -> P5 -> P2
[22:25:13] P1 -> P4 -> P5 -> P2 -> P3
[22:25:13] P1 -> P4 -> P5 -> P3 -> P2
[22:25:13] 系統處於安全狀態。
[22:25:13] 3. 處理 P1 的資源請求 (1, 1, 0, 0)：
[22:25:13] P1 請求結果: 接受: P1 請求可立即獲得，系統保持安全狀態。安全序列範例：P1 -> P4 -> P2 -> P3 -> P5
[22:25:13] 4. 處理 P4 的資源請求 (0, 0, 2, 0)：
[22:25:13] P4 請求結果: 拒絕: P4 請求資源 (0, 0, 2, 0) 超過其所需 (Need: (0, 1, 1, 2))。
```

Page Replacement Algorithms 問題陳述

- 實作並比較三種頁面置換演算法：FIFO、LRU 和 Optimal。

- 核心目標是模擬作業系統如何在物理記憶體（頁框）不足時，選擇要替換的頁面，以減少分頁錯誤 (Page Fault)。



- **虛擬記憶體 (Virtual Memory)**： 允許程式使用比實際物理記憶體更大的定址空間。

- **分頁錯誤 (Page Fault)**： 當程式存取一個不在物理記憶體中的頁面時發生，需要從輔助儲存載入該頁。

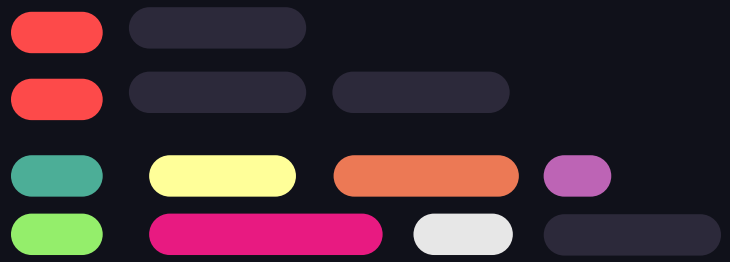
- **需求分頁 (Demand Paging)**： 頁面只在實際需要時才載入記憶體。

- **頁面置換 (Page Replacement)**： 當發生分頁錯誤且所有物理頁框都被佔用時，系統必須選擇一個現有頁面移出記憶體，為新頁面騰出空間。

Key Concept

 # 亂數頁面參考串：
- 程式首先生成一個**亂數**的頁面參考串，頁碼範圍從 0 到 9。
- 所有三種演算法都將使用**這同一串**參考串進行測試。

 # 測試條件：
- 演算法將針對頁框數量從 **1 到 7** 分別執行，並記錄每種情況下的分頁錯誤數。



 # FIFO (First-In, First-Out)：先進先出置換演算法
- **邏輯：** 替換在記憶體中**駐留時間最久**的頁面(最早進入記憶體的頁面)。
- **實作：** 使用佇列 (Queue) 追蹤頁面載入順序。

 # LRU (Least Recently Used)：最少最近使用置換演算法
- **邏輯：** 替換**最長時間未被使用**的頁面。
- **實作：** 追蹤頁面的最後存取時間或維護一個使用順序列表，最近存取的頁面會被移到列表的一端。

 # Optimal (最佳)：最佳頁面置換演算法
- **邏輯：** 替換在**未來最久時間內不會被使用**的頁面。
- **實作：** 需要「預知未來」，在參考串中向後查找頁面的下一次使用。這是**理論上的最佳演算法**，用於比較性能基準。

畫面展示

使用者可以調整參考串長度、最大頁碼、最小級最大頁框數。

參考串為執行模擬後的隨機排列的串列，並列出每種演算法在每個頁框數量下，計算並記錄所產生的分頁錯誤總數。

詳細執行日誌則能夠調整演算法類別以及頁框數來觀看演算法的執行過程。