作業系統：Synchronization Tools

**目錄**

[作業系統：Synchronization Tools 1](#_Toc204163340)

[1. Background 1](#_Toc204163341)

[1.1 範例：Producer & Consumer 問題 2](#_Toc204163342)

[2. Race Condition 3](#_Toc204163343)

[3. 臨界區問題(The Critical-Section Problem) 3](#_Toc204163344)

[4. Critical Section三大同步條件 4](#_Toc204163345)

[5. Peterson’ s Solution(彼得森解法) 4](#_Toc204163346)

[6. Hardware Support for Synchronization(硬體支援的同步工具) 5](#_Toc204163347)

[7. Memory Barriers(記憶體障壁) 5](#_Toc204163348)

[7.1 記憶體模型(Memory Model) 6](#_Toc204163349)

[8. Hardware Instructions(硬體原子指令) 6](#_Toc204163350)

[8.1 test-and-set(測試並設定) 6](#_Toc204163351)

[8.2 compare-and-swap(比較並交換，CAS) 7](#_Toc204163352)

[9. Mutex Locks(互斥鎖) 7](#_Toc204163353)

[9.1 Contention 8](#_Toc204163354)

[9.2 Spinlock(自旋鎖) 8](#_Toc204163355)

[10. Semaphore 9](#_Toc204163356)

[10.1 改進版Semaphore： 10](#_Toc204163357)

[10.2 Semaphore 相關應用：binary semaphore 10](#_Toc204163358)

[10.3 Semaphore相關應用：控制資源數量(counting semaphore) 10](#_Toc204163359)

[10.4 Semaphore相關應用：順序控制 11](#_Toc204163360)

[11. Monitor 11](#_Toc204163361)

[11.1 signal and wait vs signal and continue 12](#_Toc204163362)

[12. 存活性(Liveness) 13](#_Toc204163363)

[12.1 兩種最常見的 Liveness 問題： 13](#_Toc204163364)

[13. Bounded-Buffer Problem： 14](#_Toc204163365)

[13.1 生產者-消費者問題(Producer–Consumer Problem) 14](#_Toc204163366)

[13.2 讀者寫者問題(Readers–Writers Problem) 15](#_Toc204163367)

[13.3 哲學家進餐問題(Dining Philosophers Problem)(應該要放到章節deadlock) 16](#_Toc204163368)

# Background

在現代作業系統中，processes 和 threads 通常並發(concurrent)或平行(parallel)執行。

* 並發：一顆 CPU 上切換多個 process(快速 context switch)
* 平行：多核心同時執行多個 process

若多個 process 同時存取「共享資料」，但並未加以控制，可能導致資料錯誤。

## 範例：Producer & Consumer 問題

這段程式碼在「單執行緒」下是沒問題的，但在兩個 process 同時執行時，可能會出錯。

// Producer

while (true) {

while (count == BUFFER\_SIZE) ; // 緩衝區滿，等待

buffer[in] = next\_produced;

in = (in + 1) % BUFFER\_SIZE;

count++; // 加一個物品

}

// Consumer

while (true) {

while (count == 0) ; // 緩衝區空，等待

next\_consumed = buffer[out];

out = (out + 1) % BUFFER\_SIZE;

count--; // 移除一個物品

}

之所以會出錯的原因，在於count++ 與 count-- 的執行其實包含多個步驟。

// count++ 可能會被 CPU 翻譯成以下三步

register1 = count // 把 count 值載入暫存器

register1 = register1 + 1 // 在暫存器加一

count = register1 // 把值寫回記憶體

// count-- 也是類似的三步

register2 = count

register2 = register2 - 1

count = register2

假設 count = 5，producer 做 count++、consumer 做 count--：

* T0：producer: `register1 = count → 5
* T1：producer: register1 = register1 + 1 → 6
* T2：consumer: register2 = count → 5
* T3：consumer: register2 = register2 - 1 → 4
* T4：producer: count = register1 → count = 6
* T5：consumer: count = register2 → count = 4 ❌ 覆蓋前面的結果

# Race Condition

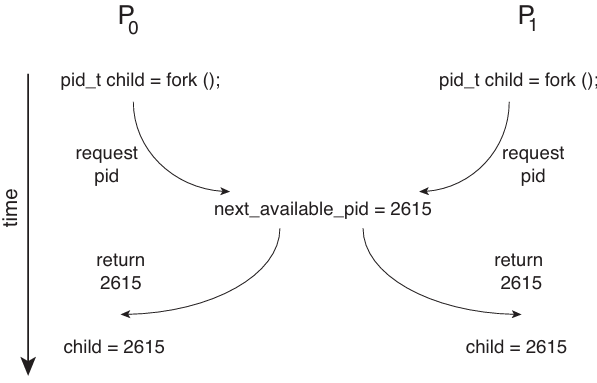
當兩個或多個 process 並發地存取與修改同一資料，便會發生Race Condition。最後的結果會決於它們交錯執行的順序。因此，我們續要同步機制(Synchronization)。

同步機制(Synchronization)意旨：在某個時間內，只有一個 process 可以修改共享變數(例如 count)。這就是我們為什麼需要臨界區(critical section)、鎖(lock)、信號量(semaphore)、mutex、monitor等等同步工具的根本原因。

此外，即使在 kernel mode，還是可能出現 race condition，像是：

* 檔案列表(open file list)：兩個 process 同時開檔案 → 都要更新 open file list → 可能造成資料錯亂
* fork() 建立新 process：假如有個共享變數next\_available\_pid(下一個可用的 process ID)，如果P0跟P1同時呼叫 `fork()`，就可能會拿到一樣的 PID。

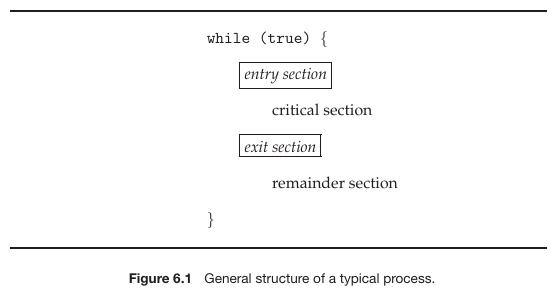
Figure 1：Race condition when assigning a pid



# 臨界區問題(The Critical-Section Problem)

當一個 process 要存取「共享資料(shared data)」時，這段會修改共享資源的程式碼就稱為 Critical Section(臨界區)。在任一時刻，最多只能有一個 process 執行在 critical section 中 => 這樣才能確保資料一致、避免 race condition。每個 process 的結構如下：

Figure 2：典型行程的一般結構



# Critical Section三大同步條件

在談同步之前，要先了解Critical Section三大同步條件：

Table 1：Critical Section三大同步條件

|  |  |
| --- | --- |
| 條件 | 說明 |
| Mutual Exclusion(互斥) | 一次只有一個 process 可以在臨界區中(防止 race condition) |
| Progress(進度) | 如果沒人處於臨界區，想進入的人必須能選出其中一人進入(不能無限拖延) |
| Bounded Waiting(有限等待) | 一個 process 等待進入臨界區的次數是有限的，不會被永遠餓著(不能 starve) |

# Peterson’ s Solution(彼得森解法)

Peterson’ s Solution(彼得森解法)是一種只用軟體就能實現兩個 process 的互斥臨界區的演算法。它是 OS 同步中的經典演算法，儘管在現代硬體上不保證正確，但它非常適合用來：

* 示範如何透過程式邏輯實現臨界區控制
* 解釋三個同步條件：Mutual Exclusion、Progress、Bounded Waiting

適用條件：僅限兩個 process(P0 與 P1)，Process 之間會輪流進入 critical section

// 共享兩個變數

int turn; // 輪到誰(0 或 1)

bool flag[2]; // flag[i] 表示 Pi 想要進入臨界區

// 演算法流程(Process Pi 的程式碼)

while (true) {

flag[i] = true; // 宣告自己要進入 CS

turn = j; // 禮讓對方先選

while (flag[j] && turn == j){} // 忍耐等待

// 🔐 Critical Section

...

flag[i] = false; // 我執行完畢，退出 CS

// 🧘 Remainder Section

}

// 模擬狀況

state 初始：flag[0] = flag[1] = false

假設P0想進入：

flag[0] = true

turn = 1

假設P1想進入：

flag[1] = true

turn = 0

while 條件：

P0: flag[1] && turn==1 → ❌ 不滿足(turn = 0)

P1: flag[0] && turn==0 → ✅ 卡住

➡ P0 可以進入

以三大同步條件來回顧Peterson’ s Solution(彼得森解法)：

* ✅Mutual Exclusion(互斥)：同時最多只會有一個人成功通過 `while` 條件
* ✅Progress(進度)：沒有 process 在 critical section 時，不會無限等待
* ✅Bounded Waiting(有限等待)：最多只會等一次對方執行完臨界區，不會無限 starve

# Hardware Support for Synchronization(硬體支援的同步工具)

在多核心或多執行緒環境中，若多個行程或執行緒同時存取共享變數(shared variable)，可能會產生競爭條件(race condition)，導致資料錯誤或程式邏輯異常。雖然我們可以使用像 Peterson’s Solution等軟體層級的協議來實現互斥(mutual exclusion)，這些方法在理論上可以滿足互斥、前進性、有限等待三大條件，但它們在現代多核心系統中可能會失效，原因包括：

* 編譯器最佳化 可能重排序讀寫指令
* CPU 指令重排(instruction reordering)
* 快取一致性問題：不同核心可能對共享記憶體有不同的本地快取版本，導致檢查的狀態不一致
* 缺乏真正的原子操作(atomic operation)支援

因此，在現代系統中，硬體層級的同步原語是實現安全互斥的必要條件。

# Memory Barriers(記憶體障壁)

一種 CPU 指令，用來阻止編譯器或處理器對指令順序的重新排序。它會強迫先前的讀寫操作一定要完成後，後續的操作才能開始。

// Thread 1

while (!flag)

memory\_barrier(); // 確保讀 flag 之前不會偷跑去讀 x

print x;

// Thread 2

x = 100;

memory\_barrier(); // 確保 x = 100 寫入後，flag 才會設為 true

flag = true;

## 記憶體模型(Memory Model)

不同的硬體架構定義了不同的「記憶體模型(memory consistency model)」，用來規範記憶體操作的順序性與可見性。

* Strongly Ordered：所有記憶體操作都依程式順序執行，一個處理器對記憶體的寫入會「立即」對其他處理器可見。較常見於 x86 架構。
* Weakly Ordered：CPU 允許對指令進行重新排序，寫入可能延遲被其他核心看到。常見於 ARM、PowerPC、RISC-V 等高效能處理器。

# Hardware Instructions(硬體原子指令)

現代 CPU 提供兩種常見原子操作，解決同步問題：

## test-and-set(測試並設定)

// 定義 test-and-set

bool test\_and\_set(bool\* target) {

bool rv = \*target;

\*target = true;

return rv;

}

// 使用 test-and-set：(這段操作是「原子」進行，兩個核心同時執行也不會競爭失敗)

bool lock = false;

while (test\_and\_set(&lock)) ; // busy-wait

// critical section

lock = false; // 解鎖

以三大同步條件來回顧test-and-set：

* ✅Mutual Exclusion(互斥)：test\_and\_set() 為原子操作，一次只會有一個執行緒成功取得鎖。
* ✅Progress(進度)：若沒有執行緒在臨界區，則一定會有一個執行緒成功取得鎖
* ❌Bounded Waiting(有限等待)：沒有保證每個執行緒都會在有限次內取得鎖，可能出現飢餓(starvation)，因為某些執行緒可能一直在競爭中失敗。

## compare-and-swap(比較並交換，CAS)

// 定義 CAS：

int compare\_and\_swap(int\* value, int expected, int new\_value) {

int temp = \*value;

if (\*value == expected)

\*value = new\_value;

return temp;

}

// 使用 CAS：(這段操作是「原子」進行，兩個核心同時執行也不會競爭失敗)

int lock = 0;

while (compare\_and\_swap(&lock, 0, 1) != 0) ; // busy-wait

// critical section

lock = 0; // 解鎖

以三大同步條件來回顧CAS：

* ✅Mutual Exclusion(互斥)：compare\_and\_swap() 為原子操作，保證只有一個執行緒能將 lock 設為 1。
* ✅Progress(進度)：沒有其他執行緒在臨界區時，必有一個執行緒會成功設置 lock。
* ❌Bounded Waiting(有限等待)：同樣沒有保證公平性，可能某個執行緒永遠搶不贏(饑餓)，這稱為 lock starvation。

**補充：**為何test-and-set跟CAS都無法滿足bounded waiting：

兩者都採用 busy-wait 自旋，而且沒有先來先服務(FIFO)或排隊機制。執行緒的搶鎖成功機率可能受排程與 CPU 核心切換影響，導致某些執行緒長時間得不到進入機會。如果要滿足有限等待，需額外設計waiting 陣列實現有限等待，保證每個 process 最多等待 n-1 次就可以進入 critical section。

# Mutex Locks(互斥鎖)

硬體原子操作(test-and-set、CAS)很底層、難用、不友善於應用程式開發者。因此作業系統提供「軟體層級」的工具：mutex

假設你想用影印機，但發現有人在用。你怎麼做？

你說：「好，我先去喝咖啡，等有人用完再叫我回來。」→ 就像 Mutex，你進入睡眠狀態(block)，等別人釋放鎖再喚醒你。

優點是不浪費資源(你去做別的事)，但缺點是喚醒需要時間(context switch)。

// 使用方式：

while (true) {

acquire(); // 嘗試鎖定，若失敗就等待(忙等)

// critical section // 共享資源的區域，只能同一時間一人進入

release(); // 解鎖，讓其他人能進入臨界區

// remainder section

}

// Mutex 的實作核心：

bool available = true;

void acquire() {

while (!available) ; // busy-wait

available = false;

}

void release() {

available = true;

}

// 這樣的操作若不是原子性的，就會有競爭問題。

// 實物上的acquire() 和 release()會由CAS來實作

## Contention

當很多 thread 同時搶鎖時，我們稱這種鎖為「有爭用的(contended)」

* uncontended lock：沒有人競爭，直接取得鎖
* contended lock：有人搶鎖，可能會進入等待(busy-wait 或 context switch)

## Spinlock(自旋鎖)

程式碼 while (!available); 一直重複檢查的過程，就稱呼為Spinlock。當一個執行緒無法取得鎖時，它會不斷重複嘗試(自旋)，而不會讓出 CPU 控制權(不會 context switch)。

假設你想用影印機，但發現有人在用。你怎麼做？你站在影印機門口，一直看：「好了沒？好了沒？好了沒？」。不去做其他事情，一直不停地觀看，就是Spinlock。優點是當影印機很快用完，你便立刻搶到；缺點是如果那個人很久都不出來，那對你而言就是浪費時間。

Table 2：Spinlock vs Mutex

|  |  |  |
| --- | --- | --- |
| 比較項目 | Spinlock | Mutex |
| 等待方式 | busy wait(不讓出 CPU) | block(會 context switch) |
| 效率 | 鎖定時間短時快 | 鎖定時間長時較穩定 |
| 適用場景 | 多核心、短臨界區 | 單核心、長臨界區 |
| 實作原理 | CAS / Test-and-Set + 自旋 | 系統呼叫 + 睡眠機制 |
| 資源使用 | 高(浪費 CPU) | 低(讓出 CPU) |

以三大同步條件來回顧Mutex：

* ✅Mutual Exclusion(互斥)：Mutex 的設計核心就是「互斥」，使用鎖來確保同一時間只有一行程可進入臨界區。
* ✅Progress(進度)：若沒有行程在臨界區，等待鎖的行程會依系統排程機制獲得鎖，理論上能前進。
* ❌Bounded Waiting(有限等待)：一般的 mutex 並不保證先來先服務，可能導致某些行程長期搶不到鎖(出現 starvation)；需額外設計公平機制。

# Semaphore

Semaphore 是另一種作業系統提供「軟體層級」的工具。Semaphore 是一個整數變數 + 兩個操作(wait() 和 signal())組成的同步工具。其核心原則為：透過 wait() 和 signal() 可以存取這個變數，而且這兩個操作必須原子性(atomic)。其優點為：

* 支援「資源共享」與「程序間通知」
* 可以排隊(不 busy-wait)
* 適用範圍廣：互斥、同步、資源限制都能做

相對而言，wait() 和 signal() 必須確保原子性(需要底層支援)，且要小心用法錯誤，可能死鎖

|  |  |
| --- | --- |
| wait(S) | 嘗試取得資源，若無則等待( 試著進去，沒得進就睡覺) |
| signal(S) | 表示釋放資源或完成某件事(通知外面的人可以進來) |

// 10. Semaphore 定義：

wait(S) {

while (S <= 0)

; // busy-wait → 初始版本會造成 busy-wait(一直繞圈圈等資源)，後面會改進。

S--;

}

signal(S) {

S++;

}

以三大同步條件來回顧Semaphore：

* ✅Mutual Exclusion(互斥)：Semaphore 可用來保護共享資源，只允許一個行程(或固定數量)進入臨界區。使用 binary semaphore(初始值為 1)可實現類似 mutex 的效果。
* ✅Progress(進度)：若沒有行程在臨界區，且有一個以上的行程等待資源，Semaphore 保證其中一個行程將會取得資源進入臨界區(透過signal()喚醒等待者)。
* ❌Bounded Waiting(有限等待)：傳統 Semaphore 並未保證一定有公平性，所以有可能某些行程一直得不到資源(飢餓 starvation)。但若搭配 FIFO 等待佇列 實作，就能滿足有限等待條件。

## 改進版Semaphore：

用 queue 排隊等待，先睡著，在等signal()喚醒

wait(semaphore \*S) {

S->value--;

if (S->value < 0) {

add this process to S->list;

sleep(); // 睡著，讓出 CPU

}

}

signal(semaphore \*S) {

S->value++;

if (S->value <= 0) {

P = remove a process from S->list;

wakeup(P); // 喚醒一個等待者

}

}

## Semaphore 相關應用：binary semaphore

// binary semaphore：若你有一個共享區域(critical section)只允許一人使用：

semaphore mutex = 1;

wait(mutex); // 鎖住

// critical section

signal(mutex); // 解鎖

// binary semaphore其行為就跟 mutex lock 一樣！

## Semaphore相關應用：控制資源數量(counting semaphore)

假設你有 3 台影印機，5 個人要用：

semaphore printers = 3;

wait(printers); // 有資源才可以用

// 使用影印機

signal(printers); // 用完歸還

## Semaphore相關應用：順序控制

假設你希望 P1 執行完 S1 之後，P2 才能執行 S2

semaphore sync = 0;

// P1

S1;

signal(sync); // 表示事情做完了

// P2

wait(sync); // 等待 P1 做完

S2;

# Monitor

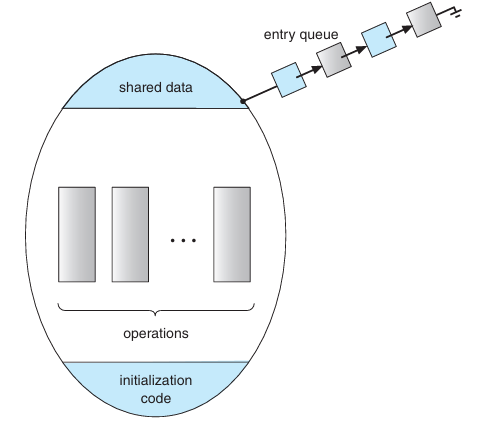
繼 mutex 和 semaphore 之後，Monitor是一種用於解決臨界區與程序同步的一種更安全、抽象化程度更高的機制。我們在回顧一下semaphore 存在的問題：

* signal()寫在wait()前面，導致多人同時進入 critical section，違反互斥
* wait()寫兩次，卡住不動造成死結
* 忘記寫signal()，鎖永遠不會釋放，其他人無法進入

Monitor 是一種「封裝資料 + 同步」的高階結構。可視為：「在高階語言裡自帶互斥鎖的同步物件」。其中：

* Monitor 是「包裹同步的空間」
* Condition 是「這個空間裡讓人等待與喚醒的機制」

Figure 3：Schematic view of a monitor



// monitor邏輯碼

monitor MonitorName {

// 共享資料

// 只允許一人進入的函數(自帶鎖)

void function1(...) { ... } // 自動加鎖

void function2(...) { ... } // 自動加鎖

initialization() { ... }

}

// 而Condition變數則用於等待與通知。

condition x;

x.wait(); // 進入等待，暫停自己

x.signal(); // 喚醒一個在等待 x 的 process

## signal and wait vs signal and continue

當 A 執行 x.signal() 喚醒 B 時，兩人不能同時待在 monitor 裡，會怎麼安排？

|  |  |
| --- | --- |
| 選項 | 說明 |
| signal-and-wait | A 喚醒 B 之後，A 自己暫停等待 |
| signal-and-continue | A 繼續做事，B 等待 A 離開 |
| (折衷)compromise | A 呼叫 signal() 後立刻離開，B 立刻進來(常見實作方式) |

以三大同步條件來回顧monitor：

* ✅Mutual Exclusion(互斥)：Monitor 保證同一時間只有一個行程能進入其內部(即使有多個條件變數)。內部程式碼區塊自動受保護。
* ✅Progress(進度)：若沒有行程在執行 Monitor 內部，則候選行程會獲得執行機會，系統能確保前進性。
* ❌Bounded Waiting(有限等待)：有些 Monitor 實作(例如 Hoare-style)可控制順序性，但多數(如 Mesa-style)不保證先來先服務，可能出現飢餓。

# 存活性(Liveness)

Liveness 意指「在並行系統中，每一個 process(或 thread)最終都能繼續往下執行，不會卡住不動。」換句話說，程式不能無限期地卡在某個地方。如果發生無限等待，會違反：

* 進度條件(Progress)
* 有界等待條件(Bounded Waiting)

## 兩種最常見的 Liveness 問題：

1. 死結(Deadlock)：所有程序彼此互相等待，沒有人能繼續執行下去。

// 若P0 拿到 S，P1 拿到 Q，接下來都等對方釋放另一個。就形成死結：兩人卡住，互等不放，永遠無法完成工作

// P0

wait(S);

wait(Q);

// P1

wait(Q);

wait(S);

1. 優先順序反轉(Priority Inversion)：高優先權的 process 被低優先權的 process 間接阻擋了，導致高優先權 process 被拖慢，甚至失敗。

範例：有三個 process：L(低)< M(中)< H(高)。L 取得了某個資源(如 lock)。H 想用這個資源，只好等 L。但 M 先跑起來，搶了 CPU 時間，讓 L 永遠沒時間釋放資源。導致 H 雖然是高優先權，卻被拖住。

解法：優先權繼承(Priority Inheritance)：L 在使用資源時，暫時繼承 H 的高優先權。這樣就能讓它優先完成、釋放資源，讓 H 順利執行。

# Bounded-Buffer Problem：

## 生產者-消費者問題(Producer–Consumer Problem)

涉及**共享資源的存取與同步問題**，因此要妥善設計互斥與同步機制。主要規則如下：

* 生產者：產生資料 → 放入緩衝區
* 消費者：從緩衝區拿出資料 → 進行消費
* 緩衝區(buffer)：有固定容量 n，不能超出範圍也不能空讀

如果沒有同步機制，會發生以下情況：

* Race Condition：生產者和消費者同時操作 buffer，造成資料錯誤或損壞
* 資料遺失或重複消費：兩個消費者同時拿同一筆資料
* Buffer overflow/underflow：資料放滿還放，或空的還拿

解法：使用 3 個 Semaphore：

|  |  |  |
| --- | --- | --- |
| Semaphore | 初始值 | 解法 |
| mutex = 1 | 保證對 buffer 的**互斥存取**(一次只有一個行程能存取) | 保證互斥 |
| empty = n | 表示 buffer 中**空格數**(初始為 n) | 控制生產節奏 |
| full = 0 | 表示 buffer 中**可用資料數**(初始為 0) | 控制消費節奏 |

// Producer 程式碼解析

while (true) {

produce\_item(); // 產生一筆資料(在臨界區外)

wait(empty); // 等待有空格可以放(同步)，確保 buffer 還有空格可以放資料。

wait(mutex); // 鎖定 buffer(互斥)，確保同一時間只有一位生產者(或消費者)能操作 buffer。

add\_to\_buffer(); // 把資料放進 buffer

signal(mutex); // 解鎖 buffer

signal(full); // 增加已用格數(通知消費者「有東西可以消費了」)

}

// Consumer 程式碼解析

while (true) {

wait(full); // 若 buffer 是空的，就等待資料出現

wait(mutex); // 鎖定 buffer(互斥)

remove\_from\_buffer(); // 取出資料

signal(mutex); // 解鎖 buffer

signal(empty); // 取走一筆資料後，告訴生產者「有空格了」

consume\_item(); // 處理資料(在臨界區外)

}

## 讀者寫者問題(Readers–Writers Problem)

在許多系統中(例如：資料庫、檔案系統)，常常會有：

* 多個「讀者」(Readers)：要查閱資料，但不會修改它。
* 一個或多個「寫者」(Writers)：要更新資料，且需要獨佔資源以確保一致性

同步規則目標如下：

* ✅允許多個讀者同時讀資料
* ❌不允許多個寫者同時寫
* ❌不允許讀者與寫者同時操作

(✅ 讀者與讀者不互斥，❌ 寫者與其他人(讀者、寫者)互斥)

解法：使用 3 個 Semaphore：

|  |  |  |
| --- | --- | --- |
| Semaphore | 初始值 | 解法 |
| read\_count | 0 | 目前有多少讀者正在讀資料 |
| mutex | 1 | 保護 read\_count 的互斥存取 |
| rw\_mutex | 1 | 控制寫者存取資料的互斥鎖，或第一個/最後一個讀者的進出權限 |

Reader 程式碼講解：

wait(mutex); // 互斥地修改 read\_count

read\_count++;

if (read\_count == 1) // 表示這是第一位讀者，為了防止寫者進入，先鎖住 rw\_mutex。

wait(rw\_mutex);

signal(mutex);

read\_data(); // 安全地讀資料(其他讀者也可能在讀)

wait(mutex); // 再次鎖住以修改 read\_count

read\_count--;

if (read\_count == 0) // 表示這是最後一位讀者離開，允許寫者進入，因此釋放 rw\_mutex

signal(rw\_mutex);

signal(mutex);

Writer程式碼講解：

wait(rw\_mutex); // 獨佔地進入臨界區(禁止讀者與其他寫者)

write\_data(); // 修改共享資料

signal(rw\_mutex); // 釋放鎖

執行示意流程：

|  |
| --- |
| 時間點 |
| R1 進入 → 檢查 read\_count = 0 → 加鎖 rw\_mutex → 開始讀 |
| R2、R3 跟著進來(因為 R1 已鎖住寫者)→ 只需改 read\_count 就好 |
| W1 嘗試進入 → 卡在 rw\_mutex(要等讀者全離開) |
| R1、R2、R3 讀完 → read\_count 減至 0 → 解鎖 rw\_mutex |
| W1 成功取得 rw\_mutex → 開始寫 |

如果一直有新的讀者進來，寫者可能永遠等不到(starvation)。若想讓寫者也有公平權，可以改為：寫者優先(Writer-Preference)或公平策略(如 FIFO queue)

## 哲學家進餐問題(Dining Philosophers Problem)(應該要放到章節deadlock)

有 5 位哲學家坐在圓桌旁。每位哲學家只有兩件事要做：吃飯(eat)與思考(think)。在每位哲學家之間放著一隻筷子(共 5 隻)，吃飯時需要同時取得左右兩邊的筷子。

* 筷子是 共享資源(Shared Resource)
* 每位哲學家必須同時取得兩隻筷子(左 + 右)才能吃飯
* 若 5 位哲學家同時先拿起左邊的筷子，就會卡住不動 → 產生死結(Deadlock)

// 原始程式碼邏輯(會產生死結)

semaphore chopstick[5]; // 每隻筷子是一個 semaphore，初始值為 1

// 每位哲學家的行為

while (true) {

wait(chopstick[i]); // 拿起左邊筷子

wait(chopstick[(i+1)%5]); // 拿起右邊筷子

eat(); // 吃飯

signal(chopstick[i]); // 放下左邊筷子

signal(chopstick[(i+1)%5]); // 放下右邊筷子

think(); // 思考

}

為什麼會發生死結？符合死結的四個必要條件：

|  |  |
| --- | --- |
| 條件名稱 | 說明 |
| ✅互斥(Mutual Exclusion) | 筷子一次只能被一位哲學家使用 |
| ✅保持與等待(Hold and Wait) | 每位哲學家持有一隻筷子後，等待另一隻 |
| ✅不可搶奪(No Preemption) | 筷子無法被強制搶走，只能主動釋放 |
| ✅循環等待(Circular Wait) | 每位哲學家都在等下一位釋放筷子(環形等候) |

解法：

1. 引入總體限制(最多同時吃飯人數 < 5)

設一個 semaphore room = 4;，讓最多 4 位哲學家進入「吃飯區」

semaphore room = 4;

semaphore chopstick[5];

while (true) {

wait(room); // 最多 4 位哲學家同時進入

wait(chopstick[i]);

wait(chopstick[(i+1)%5]);

eat();

signal(chopstick[i]);

signal(chopstick[(i+1)%5]);

signal(room); // 離開房間

think();

}

1. 哲學家有編號，其中一位反向拿筷子

例如第 4 位哲學家(編號為 4)先拿右手，再拿左手，打破「環形等待」

if (i != 4) {

wait(chopstick[i]); // 拿左

wait(chopstick[(i+1)%5]); // 拿右

} else {

wait(chopstick[(i+1)%5]); // 拿右

wait(chopstick[i]); // 拿左

}

1. 使用 Monitor 或條件變數來管理

複雜但彈性高，可讓哲學家「觀察」狀態是否允許吃飯(但偏複雜)。