作業系統：Synchronization Tools

**目錄**

[作業系統：Synchronization Tools 1](#_Toc204109931)

[1. Basic Concepts 1](#_Toc204109932)

# 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

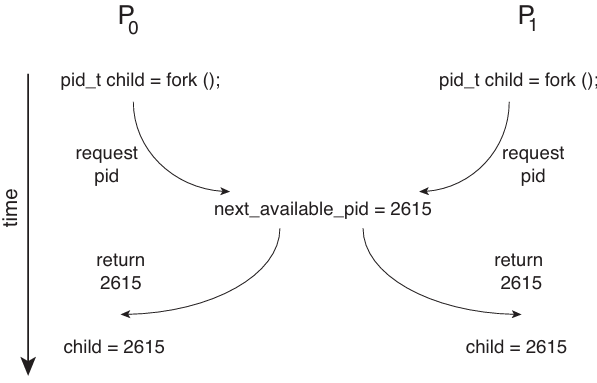
當兩個或多個 process 並發地存取與修改同一資料，便會發生Race Condition。最後的結果會決於它們交錯執行的順序。因此，我們續要同步機制（Synchronization）。

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。

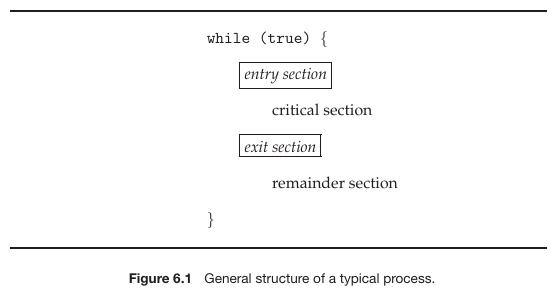
Figure 1：Race condition when assigning a pid



# 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;