# 作業系統
# Operating System

TZU-CHUN HSU[1]

[1]vm3y3rmp40719@gmail.com

[1]Department of Computer Science, Zhejiang University


2020 年 11 月 24 日
Version 2.0

# Disclaimer

本文「作業系統」為台灣研究所考試入學的「作業系統」考科使用，內容主要參考洪逸先生的作業系統參考書 [1]，以及 wjungle 網友在 PTT 論壇上提供的資料結構筆記 [2]。本文作者為 TZU-CHUN HSU，本文及其 LaTeX 相關程式碼採用 **MIT 協議**，更多內容請訪問作者之 GitHub 分頁Oscarshu0719。

# 1 Overview

1. 本文頁碼標記依照實體書 [1] 的頁碼。

2. TKB 筆記 [2] 章節頁碼：

| Chapter | Page No. | Importance |
|---|---|---|
| 1 | 3 | ⋆ ⋆ |
| 2 | 15 | ⋆ ⋆ |
| 3 | 25 | ⋆ ⋆ |
| 4 | 34 | ⋆ ⋆ ⋆ ⋆ ⋆ |
| 5 | 99 | ⋆ ⋆ ⋆ |
| 6 | 119 | ⋆ ⋆ ⋆ ⋆ ⋆ |
| 7 | 175 | ⋆ ⋆ ⋆ |
| 8 | 197 | ⋆ ⋆ ⋆ ⋆ ⋆ |
| 9 | 221 | ⋆ ⋆ ⋆ |
| 10 | 221 | ⋆ |

3. 常考：（參考實體書 [1] 中頁碼）

   (a) 113

4. 因為第六章 critical section design 部分筆記較複雜，特別分章節。

# 2 Summary

**Theorem (16)** Interrupted：

- 若頻繁 interrupt CPU 使用率仍會很低，因此若 I/O 時間不長，polling 反而可能較有利。
- External interrupt：CPU 外的周邊設備發出，例如 device controller 發出 I/O-completed。
- Internal interrupt：CPU 執行 process 遭遇重大 error，例如 Divide-by-zero，執行 priviledged instruction in user mode。
- Software interrupt：Process 需要 OS 提供服務，呼叫 system call。
- Trap：Software-generated。Catch arithmetic error，即 CPU 執行 process 遭遇重大 error，例如 Divide-by-zero。Process 需要 OS 提供服務，會先發 trap 通知 OS。

**Theorem (52)**

- Software as a service（SaaS）：Applications, e.g. Dropbox, Gmail.
- Platform as a service（PaaS）：Software stack, i.e. APIs for softwares, e.g. DB server.
- Infrastructure as a service（IaaS）：Servers or storage, e.g. storage for backup.

**Theorem (60, 61)** Process life cycles (Figure 1)：

- State：
  - New (Created)：Process is created and PCB is allocated in kernel, but memory space is NOT allocated.
  - Ready：Process is allocated memory space.
- Zombie state：Process is finished, but the parent process have NOT collected results of the children processes, or parent process have NOT executed `wait()` system call. Resources are released, but PCB have NOT been deleted, until parent process collects the results, then kernel delete PCB.

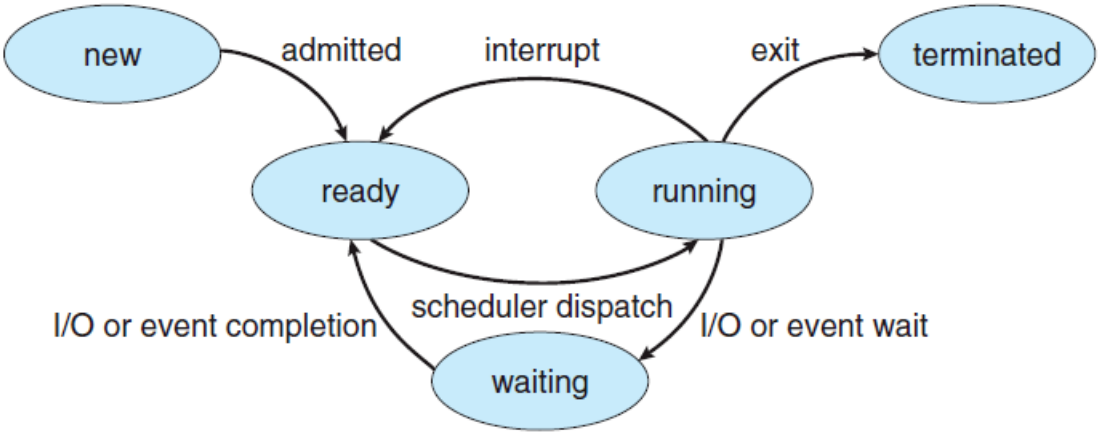


Figure 1: Process life cycles.

**Theorem (67)** Scheduler：

- Long-term (Job) scheduler：通常僅 **batch system 採用**，從 job queue 中選 jobs 載入 memory。執行頻率最低，可以調控 multiprogramming degree 與 CPU-bound 與 I/O-bound jobs 的比例。
- Short-term (CPU, process) scheduler：從 ready queue 選擇一個 process 分派給 CPU 執行。**所有系統都需要**，執行頻率最高，**無法**調控 multiprogramming degree 與 CPU-bound 與 I/O-bound jobs 的比例。
- Medium-term scheduler：Memory space 不足且有其他 processes 需要更多 memory 時執行，選擇 Blocked 或 lower priority process swap out to disk。僅 **Time-sharing system 採用**，batch 和 real-time systems 不採用，可以調控 multiprogramming degree 與 CPU-bound 與 I/O-bound jobs 的比例。

**Theorem (70)** Dispatcher：

- 將 CPU 真正分配給 CPU scheduler 選擇的 process。
- Context switch.
- Switch mode to user mode.
- Jump to execution entry of process.

**Theorem (63)** `execlp(dir, filename, args)`：載入特定工作執行，memory content 不再是 parent process 的複製，沒有參數填 `NULL`，e.g. `execlp("/bin/ls", "ls", NULL)`。

**Theorem (71, 72, 75, 76, 77, 78)** Scheduling algorithms：

- FCFS (First-come-first-serve)：可能遭遇 **Convoy effect**，即許多 processes 都在等待一個需要很長 CPU time 的 process 完成工作。
- SJF (Shortest-Job-First)：
  - Preemptive SJF 又稱 SRTF (Shortest-Remaining Time First)。
  - 效益最佳（包含 SRTF），即平均等待時間最短。
- Round-Robin (RR) scheduling：平均 turnaround time **未必**會隨著 quantum time 增加而下降。
- Multi-level queue (MQ)：易 starvation，且無法通過類似 Aging 改善。
- Multi-level feedback queue (MFQ)：可以採用類似 Aging 防止 starvation。
- Priority scheduling：

$$\begin{aligned} \text{FCFS, SJF, SRTF} &\subset \text{Priority} \\ \text{FCFS} &\subset \text{RR} \\ \text{FCFS, SJF, SRTF, Priority, RR, MQ} &\subset \text{MFQ} \end{aligned} \tag{1}$$

| Scheduling | 公平 | Preemptive | Non-preemptive | Starvation |
|---|---|---|---|---|
| FCFS | √ | | √ | |
| SJF | | | √ | √ |
| SRTF | | √ | | √ |
| Priority | | √ | √ | √ |
| RR | √ | √ | | |
| MQ | | √ | | √ |
| MFQ | | √ | | |

**Theorem (80)** Prioity inversion：Higher priority process waits for lower priority process to release resources. 通過 Priority inheritance 解決：讓 lower priority process 暫時繼承 higher priority，以便盡快取得 CPU 執行，完成後再恢復 lower priority。

**Theorem (80)** Hard real-time system：

- Schedulable 判斷：

$$\sum_{i=1}^{n} \frac{c_i}{p_i} \leq 1 \tag{2}$$

  若符合則 schedulable。其中，$n$ 為事件數目，$c_i$ 為 CPU burst time，$p_i$ 為 period time。

- Process meets deadline scheduling algorithms：
  - Rate-Monotonic：
    - Static priority 且 preemptive。
    - Period time 越小，則 priority 越高。
    - Under schedulable，也**不能**保證所有 event 皆滿足 deadline。
    - 若其無法滿足 deadline，其他 static priority scheduling 也無法。
  - EDF (Earliest Deadline First)：
    - Dynamic priority 且 preemptive。
    - Deadline 越小，則 priority 越高。
    - Under schedulable，保證所有 event 皆滿足 deadline。
    - CPU utilization 不可能達到 100%。

**Theorem (86)** Thread：

- process 是 OS 分配 **resources** 的基本單位，而 thread 是 OS 分配 **CPU time** 的基本單位。
- 同一 process 之 threads 共享 process 的 data section (static local and global variables)、heap 和 code section 等，在同一個 address 可以有多個 threads 同時執行。
- 同一 process 的不同 threads 可以平行在不同 CPUs 上執行。
- 種類：
  - User-level thread：
    - 由在 user site 的 thread library 管理，不需要 kernel 管理，e.g. POSIX 的 pthread library，但只是提供規格並沒有實現。
    - Kernel 不知道其存在，因此 kernel 不干預，導致不同 threads **無法**平行在不同 CPUs 上執行。
    - 若 user thread 發出 blocking system call 時，則該 process 也會被 blocked，即時該 process 還有其他 available threads。

  - Kernel-level thread：現行 OS 皆支持，e.g. Windows Thread library and Java Thread library。

**Theorem (88)** Thread model (user thread-to-kernel thread)：

- Many-to-One model：即 user-level thread。
- One-to-One model：
  - Not efficient than Many-to-One model.
  - e.g. Linux, family of Windows OS, OSX.
- Many-to-Many model：
  - Overhead 較 One-to-One 小，但製作較複雜。
  - NOT efficient than Many-to-One model.
  - e.g. Solaris 2，但它是用 two-level mapping model 製作，同時也允許 One-to-One。

**Theorem (136)** Deadlock：

- **必要**條件：**Mutual exclusion**, **hold and wait**, **no preemption**, and **circular wait**.
- 資源分配圖：
  - No cycle 一定 no deadlock。
  - 有 cycle 不一定 deadlock。
  - 若每一類 resources 皆 single-instance，則有 cycle 一定 deadlock。

**Theorem (139)** Deadlock prevention：

- 打破必要條件：
  - Mutual exclusion：無法破除，這是與生俱來的性質。
  - Hold and wait：除非可以一次獲得所有資源，否則不得持有任何資源；或可持有部分資源，但申請其他資源前，需先放掉所持有的所有資源。
  - No preemption：改為 preemption，但可能 starvation。
  - Circular wait：
    * 每個資源有 unique resource ID。

- process 需依照 resource ID 依序**遞增**提出申請，及持有的 resource ID，**不能大於**提出申請的 resource ID。

**Theorem (141)** Deadlock avoidance：

- Banker's algorithm：Unsafe 未必 deadlock。
- 若 $n$ processes，$m$ resources（單一種類），若滿足

$$\begin{aligned} 1 \leq Max_i \leq m \\ \sum_{i=1}^{n} Max_i < n + m \end{aligned} \tag{3}$$

則 NO deadlock。

*Proof.* 若所有資源都分配給 processes，即

$$\sum_{i=1}^{n} Allocation_i = m \tag{4}$$

又

$$\begin{aligned} \sum_{i=1}^{n} Need_i &= \sum_{i=1}^{n} Max_i - \sum_{i=1}^{n} Allocation_i \\ \rightarrow \sum_{i=1}^{n} Max_i &= \sum_{i=1}^{n} Need_i + m \end{aligned} \tag{5}$$

根據第二條件，有

$$\begin{aligned} \sum_{i=1}^{n} Max_i < n + m \\ \rightarrow \sum_{i=1}^{n} Need_i < n \end{aligned} \tag{6}$$

表示至少一個 process $P_i$，$Need_i = 0$，又

$$\begin{aligned} Max_i \geq 1 \wedge Need_i = 0 \\ \rightarrow Allocation_i \geq 1 \end{aligned} \tag{7}$$

在 $P_i$ 完工後，會產生 $\geq 1$ resources 給其他 processes 使用，又可以使 $\geq 1$ processes $P_j$ 有 $Need_j = 0$，依此類推，所有 processes 皆可完工。

**Theorem ()**

| Method | Deadlock | Utilization & throughput | Starvation | Time complexity |
|---|---|---|---|---|
| Deadlock prevention | | Low | √ | |
| Deadlock avoidance | | Low | | $O(n^2 \times m)$ |
| Deadlock detection and recovery | √ | High | | $O(n^2 \times m)$ each time |

## 2.1 Critical section design

**Theorem (170)** Critical section：

- 在 critical section，CPU 也可能被 preempted。
- 滿足：
  - Mutual exclusion：同一時間點，最多 1 process 在他的 critical section，不允許多個 processes 同時在**各自**的 critical section。
  - Progress：不想進入 critical section 時，不能阻礙其他想進入 critcal section 的 process 進入，即不能參與進入 critical section 的 decision，且必須在有限時間內決定進入 critical section 的 process。
  - Bounded waiting：Process 提出申請進入 critical section 後，必須在有限時間內進入，即公平，NO starvation。

### 2.1.1 Software support

**Theorem (171)** Two processes solution：

- Algorithm 1（錯誤）：
  - 共享變數：

```
int turn = i ∨ j;
```

Listing 1: Shared variables of Algorithm 1 (two processes solution).

  - **Progress：不成立**，當 $P_i$ 在 remainder section 且 $turn = i$，若 $P_j$ 想進入 critical section，但被 $P_i$ 阻礙，須等到 $P_i$ 進入 critcal section 再出來。

**Algorithm 1** $P_i$ of Algorithm 1 (two processes solution).

```
1: function P_i
2:     repeat
3:         while turn ≠ i do
4:         end while
5:         Critical section.
6:         turn := j
7:         Remainder section.
8:     until False
9: end function
```

- Algoroithm 2（錯誤）：
  - 共享變數：

```
// 表示是否想進入critical section。
bool flag = False;
```

Listing 2: Shared variables of Algorithm 2 (two processes solution).

  - **Progress：不成立**，當 $P_i, P_j$ 依序將 $flag := True$，在 `while` 雙方皆會等待，則 **deadlock**，皆無法進入 critcal section。

**Algorithm 2** $P_i$ of Algorithm 2 (two processes solution).

```
1: function P_i
2:     repeat
3:         flag[i] := True
4:         while flag[j] do
5:         end while
6:         Critical section.
7:         flag[i] := False
8:         Remainder section.
9:     until False
10: end function
```

- Algorithm 3 (Peterson's solution)（正確）：
  - 共享變數：

```
int turn = i ∨ j;
bool flag = False;
```

Listing 3: Shared variables of Peterson's solution (two processes solution).

- $flag$ 或 $turn$ 或兩者值皆互換依然正確，但若將前兩行賦值順序對調，則因為 **mutual exclusion 不成立**，而不正確。
- Peterson's solution is NOT guaranteed to work on modern PC, since processors and compiler may reorder read and write operations that have NO dependencies.

**Algorithm 3** $P_i$ of Peterson's solution (two processes solution).

```
1: function P_i
2:     repeat
3:         flag[i] := True
4:         turn := j
5:         while flag[j] ∧ turn = j do
6:         end while
7:         Critacal section.
8:         flag[i] := False
9:         Remainder section.
10:    until False
11: end function
```

### 2.1.2 Hardware support

**Theorem (176)** Test-and-Set:

- Test-and-Set:

```
bool Test-and-Set (bool *Target) { // Atomic.
    bool rv = *Target;
    *Target = True;
    return rv;
}
```

Listing 4: Test-and-Set.

- Algotihm 1（錯誤）:
  - 共享變數:

```
bool lock = False;
```

Listing 5: Shared variables of Algorithm 1 (Test-and-Set).

– **Bounded waiting：不成立**，可能一直領先另一個 process 搶到 CPU，導致另一個 process starvation。

**Algorithm 4** $P_i$ of Algorithm 1 (Test-and-Set).

```
1: function P_i
2:     repeat
3:         while TEST-AND-SET(&lock) do
4:         end while
5:         Critical section.
6:         lock := False
7:         Remainder section.
8:     until False
9: end function
```

- Algotihm 2（正確）：
  - 共享變數：

```
bool lock = False;
/*
True, 表示想進但在等;
False, 表示已在critical section或是初值。
*/
bool waiting[0 ··· (n − 1)] = False;
```

Listing 6: Shared variables of Algorithm 2 (TEST-AND-SET).

  - 若移除第八行 `waiting[i] := False`，則 **progress 不成立**，若僅 $P_i$ 和 $P_j$ 想進入 critical section，此時 `waiting[i], waiting[j] = True`，且 $P_i$ 先進入 critical section，有 `lock, waiting[i] = True`；當 $P_i$ 離開 critical section 後，將 `waiting[j] := False`，$P_j$ 進入 critical section；當 $P_j$ 離開 critical section 後，因為 `waiting[i] = True`，$P_j$ 將 `waiting[i] := False`，但 `lock = True`，未來沒有 process 可以再進入 critical section，**deadlock**。

**Algorithm 5** $P_i$ of Algorithm 2 (Test-and-Set).

```
1:  function P_i
2:      repeat
3:          waiting[i] := True
4:          key := True                                   ▷ Local variable.
5:          while waiting[i] ∧ key do
6:              key := TEST-AND-SET(&lock)
7:          end while
8:          waiting[i] := False
9:          Critical section.
10:         j := i + 1 (mod n)
11:         while j ≠ i ∧ ¬waiting[j] do                  ▷ 找下一個想進入的 P_j。
12:             j := j + 1 (mod n)
13:         end while
14:         if j = i then                                 ▷ 沒有 P_j 想進入 critical section。
15:             lock := False
16:         else
17:             waiting[j] := False
18:         end if
19:         Remainder section.
20:     until False
21: end function
```

**Theorem (177)** Compare-and-Swap (CAS)**:**

```
bool Compare-and-Swap (bool *value, int expected, int new_value) {
    // Atomic.
    int tmp = *value;
    if (*value == expected)
        *value = new_value;
    return tmp;
}
```

Listing 7: COMPARE-AND-SWAP.

### 2.1.3 Mutex lock

**Theorem ()** Mutex lock**:**

- OS software tools (system call).
- 共享變數**:**

```
bool Available = True;
```

Listing 8: Shared variables of Mutex lock.

**Algorithm 6** *acquire*().

```
1: function ACQUIRE
2:     while ¬Available do
3:     end while
4:     Available := False
5: end function
```

**Algorithm 7** *release*().

```
1: function RELEASE
2:     Available := True
3: end function
```

### 2.1.4 Semaphore

**Theorem (178)** Semaphore:

- OS software tools (system call).

**Algorithm 8** *wait*($S$) ($P(S)$).

```
1: function WAIT(S)                                   ▷ Atomic.
2:     while S ≤ 0 do
3:     end while
4:     S := S − 1
5: end function
```

**Algorithm 9** *signal*($S$) ($V(S)$).

```
1: function SIGNAL(S)                                 ▷ Atomic.
2:     S := S + 1
3: end function
```

**Theorem (168)** Producer-consumer problem:

- 共享變數:

```
semaphore mutex = 1;
semaphore empty = n; // buffer空格數。
semaphore full = 0; // buffer中item數。
```

Listing 9: Shared variables of Producer-consumer problem.

- 若將其中一個或兩個程式的兩行 `wait` 對調，可能會 **deadlock**。

**Algorithm 10** Producer.

```
1: function PRODUCER
2:     repeat
3:         Produce an item.
4:         WAIT(empty)
5:         WAIT(mutex)
6:         Add the item to buffer.
7:         SIGNAL(mutex)
8:         SIGNAL(full)
9:     until False
10: end function
```

**Algorithm 11** Consumer.

```
1: function CONSUMER
2:     repeat
3:         WAIT(full)
4:         WAIT(mutex)
5:         Retrieve an item from buffer.
6:         SIGNAL(mutex)
7:         SIGNAL(empty)
8:         Consume the item.
9:     until False
10: end function
```

**Theorem (182)** Reader/Writer problem：

- R/W 和 W/W 皆要互斥。
- First readers/writers problem：
  - 共享變數：

```
// R/W和W/W互斥控制，同時對writer不利之阻擋。
semaphore wrt = 1 ;
```

```
int readcnt = 0;
semaphore mutex = 1; // readcnt互斥控制。
```

Listing 10: Shared variables of First Reader/Writer problem.

**Algorithm 12** Writer (First Reader/Writer problem).

```
1: function WRITER
2:     repeat
3:         WAIT(wrt)
4:         Writing.
5:         SIGNAL(wrt)
6:     until False
7: end function
```

**Algorithm 13** Reader (First Reader/Writer problem).

```
1: function READER
2:     repeat
3:         WAIT(mutex)
4:         readcnt := readcnt + 1
5:         if readcnt = 1 then                ▷ 表示第一個 reader 需偵測有無 writer 在。
6:             WAIT(wrt)
7:         end if
8:         SIGNAL(mutex)
9:         Reading.
10:        WAIT(mutex)
11:        readcnt := readcnt − 1
12:        if readcnt = 0 then                ▷ No reader.
13:            SIGNAL(wrt)
14:        end if
15:        SIGNAL(mutex)
16:    until False
17: end function
```

- Second Reader/Writer problem：
  - 共享變數：

```
int readcnt = 0;
semaphore mutex = 1; // readcnt互斥控制。
semaphore wrt = 1; // R/W和W/W互斥控制。
int wrtcnt = 0;
```

```
semaphore y = 1; // wrtcnt互斥控制。
semaphore rsem = 1; // 對reader不利之阻擋。
semaphore z = 1; // reader的入口控制，可有可無。
```

Listing 11: Shared variables of Second Reader/Writer problem.

**Algorithm 14** Writer (Second Reader/Writer problem).

```
1: function WRITER
2:     repeat
3:         WAIT(y)
4:         wrtcnt := wrtcnt + 1
5:         if wrtcnt = 1 then                ▷ 表示第一個 writer 需阻擋 readers。
6:             WAIT(rsem)
7:         end if
8:         SIGNAL(y)
9:         WAIT(wrt)
10:        Writing.
11:        WAIT(y)
12:        wrtcnt := wrtcnt − 1
13:        if wrtcnt = 0 then
14:            SIGNAL(rsem)                  ▷ No writer.
15:        end if
16:        SIGNAL(wrt)
17:        SIGNAL(y)
18:    until False
19: end function
```

**Algorithm 15** Reader (Second Reader/Writer problem).

```
1: function READER
2:     repeat
3:         WAIT(z)
4:         WAIT(rsem)
5:         WAIT(mutex)
6:         readcnt := readcnt + 1
7:         if readcnt = 1 then
8:             WAIT(wrt)
9:         end if
10:        SIGNAL(mutex)
11:        SIGNAL(rsem)
12:        SIGNAL(z)
13:        Reading.
14:        WAIT(mutex)
15:        readcnt := readcnt − 1
16:        if readcnt = 0 then
17:            SIGNAL(wrt)
18:        end if
19:        SIGNAL(mutex)
20:    until False
21: end function
```

**Theorem (184)** The sleeping barber problem：

- 共享變數：

```
semaphore customer = 0; // 強迫barber sleep。
// 強迫customer sleep if barber is busy。
semaphore barber = 0;
int waiting = 0; // 正在等待的customers個數。
semaphore mutex = 1; // waiting互斥控制。
```

Listing 12: Shared variables of The sleeping barber problem.

- 若將 BARBER 將兩行 `wait` 對調，可能會 **deadlock**。

**Algorithm 16** Barber.

```
1: function BARBER
2:     repeat
3:         WAIT(customer)                          ▷ Barber go to sleep if no customer.
4:         WAIT(mutex)
5:         waiting := waiting − 1
6:         SIGNAL(barber)                          ▷ Wake up customer.
7:         SIGNAL(mutex)
8:         Cutting hair.
9:     until False
10: end function
```

**Algorithm 17** Customer.

```
1: function CUSTOMER
2:     repeat
3:         WAIT(mutex)
4:         if waiting < n then                     ▷ 入店。
5:             waiting := waiting + 1
6:             SIGNAL(mutex)
7:             SIGNAL(customer)                    ▷ Wake up barber.
8:             WAIT(barber)                        ▷ Customer go to sleep if barber is busy.
9:             Getting cut.
10:        else
11:            SIGNAL(mutex)
12:        end if
13:    until False
14: end function
```

**Theorem (187)** The dining-philosophers problem：

- 五位哲學家兩兩間放一根筷子吃中餐（筷子），哲學家需取得左右兩根筷子才能吃飯。若吃西餐（刀叉），必須偶數個哲學家，
- Algorithm 1（錯誤）：
  - 共享變數：

```
semaphore chopstick[0 ··· 4] = 1;
```

Listing 13: Shared variables of The dining-philosophers problem.

– 會 **deadlock**，若每位哲學家皆取左手邊的筷子，則每個哲學家皆無法拿起右手邊的筷子。

**Algorithm 18** Algorithm 1 $P_i$ (The dining-philosophers problem).

```
1: function P_i
2:     repeat
3:         Hungry.
4:         WAIT(chopstick[i])
5:         WAIT(chopstick[(i + 1 (mod 5))])
6:         Eating.
7:         SIGNAL(chopstick[i])
8:         SIGNAL(chopstick[(i + 1 (mod 5))])
9:         Thinking.
10:    until False
11: end function
```

- Algorithm 2（正確）：
  - 根據公式 (3)，人數必須 $< 5$ 才不會 deadlock。
  - 共享變數：

```
semaphore chopstick[0 ··· 4] = 1;
// 可拿筷子的哲學家數量互斥控制。
semaphore no = 4;
```

Listing 14: Shared variables of The dining-philosophers problem.

**Algorithm 19** Algorithm 2 $P_i$ (The dining-philosophers problem).

```
1: function P_i
2:     repeat
3:         WAIT(no)
4:         (Same as Algorithm 1.)
5:         SIGNAL(no)
6:     until False
7: end function
```

- Algorithm 3：只有能夠同時拿左右兩根筷子才允許持有筷子，否則不可持有任何筷子，**破除 hold and wait**，不會 deadlock。
- Algorithm 4：當有偶數個哲學家時，偶數號的哲學家先取左邊，再取右邊，奇數號的則反之，**破除 circular wait**，不會 deadlock。與吃西餐先拿刀再拿叉相似。

**Theorem ()** Binary semaphore 製作 counting semaphore（若為 $-n$ 表示 $n$ 個 process 卡在 `wait`）：

- 共享變數：

```
int c = n; // Counting semaphore號誌值。
semaphore s1 = 1; // c互斥控制。
binary_semaphore s2 = 0; // c < 0時卡住process
```

Listing 15: Shared variables of The dining-philosophers problem.

**Algorithm 20** *wait*($c$) (counting semaphore).

```
1: function WAIT(c)
2:     WAIT(s1)
3:     c := c − 1
4:     if c < 0 then
5:         SIGNAL(s1)
6:         WAIT(s2)                ▷ Process 卡住。
7:     else
8:         SIGNAL(s1)
9:     end if
10: end function
```

**Algorithm 21** *signal*($c$) (counting semaphore).

```
1: function SIGNAL(c)
2:     WAIT(s1)
3:     c := c + 1
4:     if c ≤ 0 then               ▷ 先前有 process 卡住。
5:         SIGNAL(s2)
6:     end if
7:     SIGNAL(s1)
8: end function
```

**Theorem ()** Non-busy waiting semaphore：

```
struct semaphore {
    int value;
    Queue Q; // Waiting queue.
}
```

Listing 16: Non-busy waiting semaphore.

**Algorithm 22** *wait*($S$) (non-busy waiting semaphore).

```
1: function WAIT(S)
2:     S.value := S.value − 1
3:     if S.value < 0 then
4:         Add process p into S.Q.
5:         block(p)    ▷ System call 將 p 的 state 從 running 改為 wait，有 context switch cost。
6:     end if
7: end function
```

**Algorithm 23** *signal*($S$) (non-busy waiting semaphore).

```
1: function SIGNAL(S)
2:     S.value := S.value + 1
3:     if S.value ≤ 0 then
4:         Remove process p from S.Q.
5:         wakeup(p)    ▷ System call 將 p 的 state 從 wait 改為 ready，有 context switch cost。
6:     end if
7: end function
```

**Theorem ()** 製作 semaphore：

- Algorithm 1 (disable interrupt and non-busy waiting)：

**Algorithm 24** *wait*($S$) of Algorithm 1 (disable interrupt and non-busy waiting).

```
1: function WAIT(S)
2:     Disable interrupt.
3:     S.value := S.value − 1
4:     if S.value < 0 then
5:         Add process p into S.Q.
6:         Enable interrupt.
7:         block(p)
8:     else
9:         Disable interrupt.
10:    end if
11: end function
```

**Algorithm 25** *signal*$(S)$ of Algorithm 1 (disable interrupt and non-busy waiting).

```
1: function SIGNAL(S)
2:     Disable interrupt.
3:     S.value := S.value + 1
4:     if S.value ≤ 0 then
5:         Remove process p from S.Q.
6:         wakeup(p)    ▷ System call 將 p 的 state 從 wait 改為 ready，有 context switch cost。
7:     end if
8:     Enable interrupt.
9: end function
```

- Algorithm 2 (critacal section design and non-busy waiting)：將 Algorithm 1 (2.1.4) 中的 `Enable interrupt.` 和 `Disable interrupt.` 分別改為 `Entry section.` 和 `Exit section.` 並使用 Test-and-Set (5) 或 Compare-and-Swap (7) 實現。

- Algorithm 3 (disable interrupt design and busy waiting)：

**Algorithm 26** *wait*$(S)$ of Algorithm 3 (disable interrupt design and busy waiting).

```
1: function WAIT(S)
2:     Disable interrupt.
3:     while S ≤ 0 do
4:         Enable interrupt.
5:         Disable interrupt.
6:     end while
7:     S := S − 1
8:     Enable interrupt.
9: end function
```

**Algorithm 27** *signal*$(S)$ of Algorithm 3 (disable interrupt design and busy waiting).

```
1: function SIGNAL(S)
2:     Disable interrupt.
3:     S := S + 1
4:     Enable interrupt.
5: end function
```

- Algorithm 4 (critical section design and busy waiting)：同 Algorithm 2 (2.1.4)，將 Algorithm 3 (2.1.4) 中的 `Enable interrupt.` 和 `Disable interrupt.` 分別改為 `Entry section.` 和 `Exit section.` 。

### 2.1.5 Monitor

**Theorem (189)** Monitor：

- High-level abstraction, belongs to programming language.
- Condition variables：在 monitor 內的 condition type 變數，用以解決 synchronization problem。提供 member functions：
  - `wait`：Block process 放到各自的 waiting queue。
  - `signal`：若有 process 在其 waiting queue 則恢復該 process 執行，否則無作用。
  - `wait(c)`：Conditional wait，有時候需要 priority queue 包含 entry 和 waiting queue，priority 越高越先移出，其中 $c$ 表示 priority number。
- Process is NOT active：
  - Process 呼叫的 function 執行完畢。
  - Process 執行 `wait()` 被 blocked。

**Theorem (191)** Monitor 解 The dining philosophers problem：

```
Monitor Dining-ph {
    enum {
        thinking, hungry, eating
    } state[5];
}
Condition self[5];
```

Listing 17: Data structure (The dining philosophers problem (Monitor)).

**Algorithm 28** *pickup*($i$).

```
1: function PICKUP(i)
2:     state[i] := hungry
3:     TEST(i)
4:     if state[i] ≠ eating then
5:         self[i].WAIT
6:     end if
7: end function
```

**Algorithm 29** *test*$(i)$.

1: **function** TEST$(i)$
2: **if** $state[(i+4) \pmod 5] \neq eating \wedge state[i] = hungry \wedge state[(i+1) \pmod 5] \neq eating$ **then**
3: $state[i] := eating$
4: $self[i]$.SIGNAL
5: **end if**
6: **end function**

**Algorithm 30** *putdown*$(i)$.

1: **function** PUTDOWN$(i)$
2: $state[i] := thinking$
3: TEST$((i+4) \pmod 5)$
4: TEST$((i+1) \pmod 5)$
5: **end function**

**Algorithm 31** *initialization_code*$(i)$.

1: **function** INITIALIZATION_CODE ▷ For non-Condition type.
2: **for** $i := 0$ to $4$ **do**
3: $state[i] := thinking$
4: **end for**
5: **end function**

**Algorithm 32** $P_i$ (The dining philosophers problem (Monitor)).

1: **function** $P_i$
2: Dining_ph dp ▷ Shared variable.
3: **repeat**
4: Hungry. ▷ No active.
5: $dp$.PICKUP$(i)$ ▷ Running: active; Blocked: NOT active.
6: Eating. ▷ No active.
7: $dp$.PUTDOWN$(i)$ ▷ Active.
8: Thinking. ▷ No active.
9: **until** $False$
10: **end function**

**Theorem ()** Example of monitor：若有三台 printers，且 process ID 越小，priority 越高。

```
Monitor PrinterAllocation {
    bool pr[3];
    Condition x;
```

```
    }
```

Listing 18: Data structure of example of monitor

**Algorithm 33** *Apply*($i$).

```
1: function APPLY(i)
2:     if pr[0] ∧ pr[1] ∧ pr[2] then
3:         x.WAIT(i)
4:     else
5:         n := Non-busy printer
6:         pr[n] := True
7:         return n
8:     end if
9: end function
```

**Algorithm 34** *Release*().

```
1: function RELEASE(n)
2:     pr[n] := False
3:     x.SIGNAL
4: end function
```

**Algorithm 35** *initialization_code*().

```
1: function INITIALIZATION_CODE
2:     for i := 0 to 2 do
3:         pr[i] := False
4:     end for
5: end function
```

**Algorithm 36** $P_i$ of example of monitor.

```
1: function P_i
2:     pa := PRINTERALLOCATION                ▷ Shared variable.
3:     n := pa.APPLY(i)
4:     Using printer pr[n] .
5:     pa.RELEASE(n)
6: end function
```

**Theorem ()** 使用 Monitor 製作 binary semaphore：

```
    Monitor Semaphore {
        int value;
```

```
        Condition x;
    }
```

Listing 19: Data structure (Making semaphore using monitor).

**Algorithm 37** *Wait*().

1: **function** WAIT
2: **if** $value \leq 0$ **then**
3: $x$.WAIT
4: **end if**
5: $value := value - 1$
6: **end function**

**Algorithm 38** *Signal*().

1: **function** SIGNAL
2: $value := value + 1$
3: $x$.SIGNAL
4: **end function**

**Algorithm 39** *initialization_code*().

1: **function** INITIALIZATION_CODE
2: $value := 1$
3: **end function**

**Theorem ()** 使用 semaphore 製作 monitor：

- 共享變數：

```
semaphore mutex = 1;
// Block process P if P call signal.
semaphore next = 0;
// 統計process P 那種特殊waiting processes的個數。
int next_cnt = 0;
// Block process Q if Q call wait.
semaphore x_sem = 0;
// 統計一般waiting processes的個數。
int x_cnt = 0;
```

Listing 20: Shared variables of making monitor using semaphore.

- 在 function body 前後加入控制碼，類似 Entry section 和 Exit section。

**Algorithm 40** $f$ (Example for adding control code before and after function body).

```
1: function F
2:     WAIT(mutex)
3:     Function body.
4:     if next_cnt > 0 then
5:         SIGNAL(next)
6:     else
7:         SIGNAL(mutex)
8:     end if
9: end function
```

**Algorithm 41** $x.wait()$.

```
1: function x.WAIT
2:     x_cnt := x_cnt + 1
3:     if next_cnt > 0 then
4:         SIGNAL(next)
5:     else
6:         SIGNAL(mutex)
7:     end if
8:     WAIT(x_sem)                    ▷ Q 自己卡住。
9:     x_cnt := x_cnt − 1             ▷ Q 被救。
10: end function
```

**Algorithm 42** $x.signal()$.

```
1: function x.SIGNAL
2:     if x_cnt > 0 then
3:         next_cnt := next_cnt + 1
4:         SIGNAL(x_sem)
5:         WAIT(next)                 ▷ P 自己卡住。
6:         next_cnt := next_cnt − 1   ▷ P 被救。
7:     end if
8: end function
```

**Theorem (229)** TLB 使用時，Effective Memory Access Time (EMAT)：

$$\text{EMAT\_TLB} = \text{TLB time} + (2 - p) \times \text{Memory access time} \tag{8}$$

$p$ is TLB hit rate.

**Theorem (242)** Virtual memory：

- Less I/O time，但總體 I/O time 上升，因為次數提升。
- $$\text{EMAT} = (1-p) \times \text{Memory Access time} + p \times \text{Page fault time} \tag{9}$$

  $p$ is page fault rate.

**Theorem (246)** Page replacement：

- LRU 近似：
  - Second chance (Clock)：先用 FIFO 挑出一個 page，若同時 reference bit 為 0，則為 victim page，但若為 1，則 reset reference bit，loading time 改為現在時間，重新 FIFO 找 page。
  - Enchanced second chance：選擇 $< reference, dirty >$ 最小者，若多個 pages 相同，則採用 FIFO。
- 所有 replacement algorithms 沒有最差，只有最佳。
- Belady amonaly：分給 process 的 frames 增加，但 page fualt rate 不降反升。
- Stack property：$n$ frames 所包含的 page set 必定是 $n+1$ frames 所包含的 page set 的子集合。且具有 stack property 的法則，不會發生 belady anomaly。只有 **OPT** 和 **LRU** 有。
- Page buffering：
  - Free frames pool：將 frames 分為
    - Resident frames 分配給 process。
    - Free frames pool，OS keep，讓 miss pages 先行載入，process 即可恢復執行，且加入 resident frames，完成後再將 victim page write back to disk，並歸還 free frames pool。
  - Keep modification list 紀錄 dirty bit 為 1 的所有 page no.，等到 OS 空閒再將 list 中的 pages write back to disk and reset dirty bits。

**Theorem (253)** Frame allocation：

- Process 可分配 frames 數量由 hardware 決定，最多為 physical memory size，最少須讓任一 machine code 完成，即週期中最多可能 memory access 數量，e.g. $IF, MEM, WB$ 共三次。

- 解決 thrashing：
  - 若已發生，只能降低 multiprogramming degree。
  - 利用 page fault frequency control 防止 thrashing，OS 訂定 page fault rate 合理的上下限，thrashing 理當不會發生。若大於上限，應該多分配 frames；若小於下限，應該取走一些 frames。
- Working-set model：
  - 若符合 locality，則可降低 page fault rate；違反 locality：linked list, hashing, binary search, jump, indirect addressing mode。
  - 可預防 thrashing，對於 prepaging 也有益。
- 
  - Bigger paging disk 沒幫助。
  - **Faster paging disk 有幫助**，因為 decrease page fault time。
  - **Increase page size 有幫助**。
  - Decrease page size 沒幫助。
  - **Local repalcement 有幫助**，因為 thrashing 不至於擴散。
  - **Prepaging 有幫助**，若猜測夠準，decrease page fault rate。

**Theorem (258)** Page size 越小：

- Page fault rate 增加。
- Page table size 增加。
- I/O 次數增加。
- Internal fragmentation 輕微。
- I/O transfer time 較小。
- Locality 較佳。
- 趨勢：大 page size。

**Theorem (261)** Copy-on-write：

- `fork()` without Copy-on-write：Child and parent processes 占用不同空間，且複製 parent process content 給 child process，大幅增加 memory space 需求，且 process creation 較慢。
- `fork()` with Copy-on-write：Child process 共享 parent process memory space，不 allocate new frames，降低 memory space 需求，且不須複製 parent process content，process creation 較快，但在 write 時，allocate new frame 給 child process，並複製內容，修改 child process 的 page table 指向 new frame，最後才 write。
- `vfork()` (Virtual memory `fork()`)：Child process 共享 parent process memory space，不 allocate new frames，但不提供 Copy-on-write，因此在 write 時，另一方會受影響。適用於 child process create 後馬上 `execlp()`，e.g. UNIX shell command interpreter。

**Theorem ()** Seek time：Head 移到 **track** time，通常最長；Latency (Rotation) time：Head 移到 **sector** time。

**Theorem (304)** File allocation methods：

- FAT (File Allocation Table)：Linked allocation，將 linking info 存在 FAT (memory) 中，不存在 allocation disk，加速 access。
- UNIX 的 I-Node：Index allocation，對所有大小檔案皆適合。
- Internal fragmentation problem 所有 allocation 皆有，可以視而不見。

**Theorem (309)** Disk scheduling：

- FCFS：效果不好。
- SSTF (Shortest Seek-time Track First)：效果不錯。
- SCAN：效果尚可，適用大量 tracks request，獲得較均勻等待時間，不盡公平，但 NO starvation。
- 無最差也無最佳。

| Scheduling | Starvation |
|---|---|
| FCFS | |
| SSTF | √ |
| SCAN | |
| C-SCAN | |
| Look | |
| C-Look | |

# References

[1] 洪逸. 作業系統金寶書（含系統程式）. 鼎茂圖書出版股份有限公司, 4 edition, 2019.

[2] wjungle@ptt. 作業系統 @tkb 筆記. https://drive.google.com/file/d/0B8-2o6L73Q2VelFZaXpBVGx2aWM/view?usp=sharing, 2017.