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 21 日
Version 1.0

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 (7) Spooling (Simultaneous Peripheral Operations On-Line processing)：將 disk 當作 buffer，將輸出檔先儲存在 disk，直到 device 取走，達到多個 I/O devices 的效果。

Theorem (10) Real-time system：

- 類型：
  - Hard real-time system：
    - 若工作未在 deadline 前完成，即為失敗。
    - 處理時間過長或無法預測的設備少用，disk 少用，virtual memory 不用。
    - 不與 time-sharing system 並存。
    - 減少 kernel 介入時間。
  - Soft real-time sharing：
    - real-time processes 必須持續保有最高 priority 直到結束。
    - 必須支援 preemptive priority scheduling。
    - 不提供 Aging。
    - 減少 kernel 的 dispatch latency。
    - 可與 virtual memory 和 time-sharing 共存，但是 real-time processes 不能被 swapped out 直到完工。
    - 現行 OS 皆支援。

Theorem (16) Interrupted：

- CPU 親自監督 I/O 完成。
- 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。
- 分類：
  - 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。

- Interrupt 要分 priority。
  - Non-maskable interrupt：通常是重大 error 引起，需要立即處理，例如 internal interrupt。
  - Maskable interrupt：可以忽略或是延後處理，例如 software interrupt。

Theorem (17) DMA (Direct Memory Access)：

- DMA controller 代替 CPU 負責 I/O 與 memory 間的傳輸。
- 適用 block-transfer，interrupt 頻率較低，一個 block 才中斷一次，但設計較複雜。
- 造成 resource contention（IF、MEM 和 WB 週期），因此採用 interleaving (cycle stealing)，與 CPU 輪流使用 memory 與 bus。
- DMA 比 CPU 有更高 priority (SJF)。

Theorem (18) I/O：

- Non-blocking I/O 與 Asynchronous I/O 差異：前者會有多少通知多少，後者會等 I/O 全部完成才通知 process。
- 種類：
  - Memory-mapped I/O：無專門 I/O 指令，但特別分一塊 memory 給 I/O，寫入該塊 memory 視為 I/O 操作。
  - Isolated I/O：有專門 I/O 指令。

Theorem (34) Virtual machine：

- Host：Underlying hardware system.
- Virtual Machine Manager (VMM, Hypervisor).
- Guest：Process provided by VM, for example, OS, applications.
- 不易開發 VMM，因為要複製與底層 host hardware 一模一樣的 VM 非常困難，例如 modes control and transition、資源調度和 I/O device and controller 之模擬。
- 效能比 host hardware 差。
- Implementations：

  - Type 0：Hardware.
  - Type 1：Kernel mode.
    - OS-like software：只提供 virtualization。
    - General-purpose OS 在 kernel mode 提供 VMM services。
  - Type 2：User mode：Applications that provide VMM services.
- Virtualization variations：
  - Paravirtualization：Present guest similar but NOT identical to host hardware. Guest should be modified to run on paravirtualized hardware.
  - Emulators：Applications to run on a different hardware environment.
  - Application containment (container)：Not virtualization at all, but provides segregating applications from the OS.
- Java Virtual Machine (JVM)：只提供規格（class loader，class verifier 和 Java interpreter），並非實現。

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.
- Transition：
  - Admitted：Process is allocated memory space. In batch system, use long-term scheduler to decide which process to load into memory. In time-sharing and real-time systems, do NOT use long-term scheduler.

  - Dispatch：Short-term (CPU) scheduler decides which process to get CPU allocation and allocates CPU to execute.
  - Interrupt (Time-out)：e.g. time-out interrupt.

- 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.

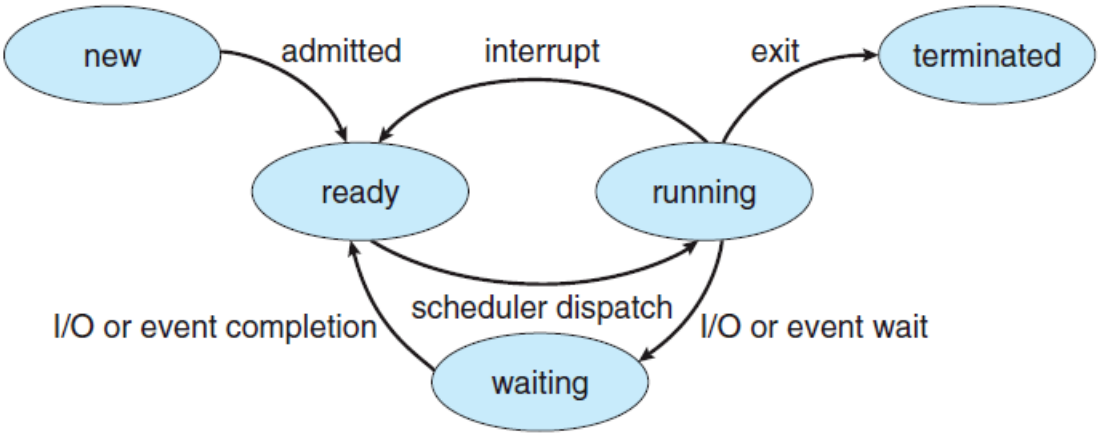

Figure 1: Process life cycles.

Theorem (67) Scheduler：

- Long-term (Job) scheduler：通常 batch system 採用，time-sharing 與 real-time systems 不採用，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。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 (69, 70)

- Context switch：
  - 執行期間無法執行 process，主要取決於硬體因素。
  - 降低負擔：
    - 提供 Multiple registers set：每個 process 有自己的 registers set，只需要切換 pointer 就能 context switch。
    - 使用 Multi-threading。
- 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) Process control operation：

- `fork()`：
  - child process 有與 parent process 不同的 memory space，而起始 code section 和 data section 皆來自 parent process 的複製。
  - 失敗：回傳負值；成功：回傳 0 給 child process，> 0 值即 child process PID 給 parent process。
- `wait()`：若 child process 已終止，帶 parent process 還沒執行 $wait()$，但 kernel 含不能清除 child process PCB，直到 parent process 收集完 child process info，此段期間稱 zombie。
- `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 (70) 評估 scheduling performance：

- Waiting time：Process 在 **ready queue** 的時間。
- Turnaround：Process **進入系統到完成工作**的時間。
- Response time：User 輸入到系統產生第一個回應的時間。

Theorem (75) Starvation：

- 有機會完成但是機會很小。
- 通過 Aging 解決：當 process 在系統中時間增加，逐漸提升 process 的 prioirty。但是 soft real-time systems 不能使用 Aging，因為會違反定義。

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。
  - 對 long-burst time job，可能有 starvation。
  - 不適合用在 CPU (short-term) scheduling，因為不知道 process 的精確 next CPU burst time，短時間也難以精準預估。
  - Long-term job 可能可行。
  - Exponential Average：預估 next CPU burst time。

    $$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n \tag{1}$$

    其中，t_n 為本次 CPU 實際值，α 為機率。
- Round-Robin (RR) scheduling：
  - Time-sharing system 採用。
  - 效能取決於 time quantum 大小，太小 context switch 太頻繁。
  - Time quantum 大小會影響 turnaround time，平均 turnaround time **未必**會隨著 quantum time 增加而下降。
- Multi-level queue (MQ)：
  - Queue 之間採用 fixed priority preemptive scheduling 或 RR。
  - 每個 queue 有各自的 scheduling。
  - 不允許 process 在 queues 間移動，因此缺乏彈性。

  - 易 starvation，且無法通過類似 Aging 改善。

- Multi-level feedback queue (MFQ)：
  - 允許 process 在 queues 間移動，可降級增加彈性。
  - No starvation，可以採用 Aging 防止 starvation。

- Priority scheduling：

$$\begin{aligned} \text{FCFS, SJF, SRTF} &\subset \text{Priority} \\ \text{FCFS} &\subset \text{RR} \\ \text{FCFS, SJF, SRTF, Priority, RR, MQ} &\subset \text{MFQ} \end{aligned} \tag{2}$$

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

Theorem (79) Multiprocessors system：

-
  - ASMP：類似 single-CPU scheduling。
  - SMP：
    - 所有 CPU 共用一條 ready queue，no load balancing problem，須防止 race condition。
    - 每一個 CPU 有各自 ready queue，可能有 load balancing problem，可以通過 kernel 協調 imbalance CPUs 給其他 ideal CPUs processes。

- Processor affinity：
  - 一但 process 在某 CPU 上執行，盡量避免 migration。
  - 因為 migration from one CPU to another，first CPU cache should be invalidated and second CPU cache should be repopulated, and it costs a lot。

  - Migrating a process may incur a penalty on NUMA systems, where a process may be moved to a CPU that requires longer memory access time。

- Multicores：有 Memory stall problem。
  - 通過 Multithreaoded processing cores 解決，2 or more hardware threads are assigned to each core。
  - 當一 thread memory stall，core can switch to another thread。
  - 每個 thread OS 視為 logical CPU，皆可執行 software thread (process)，稱為 Cheap Multithreading Technology (CMT)。

Theorem (80) Real-time system：

- Soft real-time system：
  - 採用 preemptive 和 priority scheduling，不提供 aging。
  - Minimize latency：包含 interrupt latency 和 dispatch latency (Figure 2)，real-time system 不適合 non-preemptive；若是 preemptive，須防止 race condition。
  - 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。

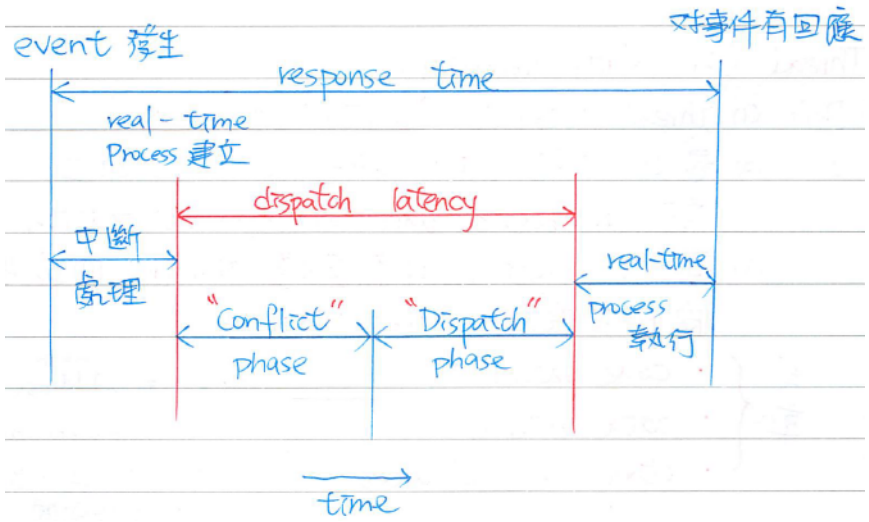

Figure 2: Dispatch latency on real-time systems.

Theorem (80) Hard real-time system：

- 討論 Synchronous real-time event scheduling，即每隔一段時間發生，需在 deadline 前完成。
- Schedulable 判斷：

$$\sum_{i=1}^{n} \frac{c_i}{p_i} \leq 1 \tag{3}$$

  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：

- 又稱 Lightweight process。
- 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。
- 若一 thread 被 blocked，則 CPU 可切換給其他 thread 執行，所以 process 不會 blocked。
- Private contents 較 traditional prcocess 少，context switch 較快。
- 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。

- pthread library is provided by user-level or kernel-level.
- Windows Thread library and Java Thread library are kernel-level threads.

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.
  - 若建立過多 user thread，kernel overhead 過重，performance 下降，一般會限制 thread 數量。
  - 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 (90) Threading issues：

- `fork()` issue：若 parent 和 child thread 工作相同，則複製 parent process 所有 threads 到 child thread；反之，則只複製該 thread 到 child process。
- Signal delivery issue：
  - 種類：

    - Synchronous：自作自受，e.g. Divide-by-zero。
    - Asynchronous：池魚之殃，e.g. aborted by system admin, parent 被終止，child 也一同被終止。
  - Signal delivery：
    - Deliver signal to the thread to which the signal applies, e.g. synchronous signal.
    - Deliver signals to all threads, e.g. user abortion.
    - Deliver signals to some threads, e.g. kill.
    - Assign a specific thread to receive all signals for its process, e.g. Solaris 2.

Theorem (136) Deadlock：

- **必要**條件：Mutual exclusion, hold and wait, no preemption, 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

$$\begin{aligned} 1 \leq Max_i \leq m \\ \sum_{i=1}^{n} Max_i < n + m \end{aligned} \tag{4}$$

則 NO deadlock。

Proof. 若所有資源都分配給 processes，即

$$\sum_{i=1}^{n} Allocation_i = m \tag{5}$$

又

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

根據第二條件，有

$$\begin{aligned} \sum_{i=1}^{n} Max_i < n + m \\ \rightarrow \sum_{i=1}^{n} Need_i < n \end{aligned} \tag{7}$$

表示至少一個 process P_i，$Need_i = 0$，又

$$\begin{aligned} Max_i \geq 1 \wedge Need_i = 0 \\ \rightarrow Allocation_i \geq 1 \end{aligned} \tag{8}$$

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

- 若所有類型資源皆為 single-instance，可用畫圖簡化 procedure。
  - claim edge：未來 process P_i 會對 resource R_j 提出申請。
  - Procedures：
    - 檢查是否有 claim edge，若無終止 P_i。
    - 檢查 R_j 是否 available，若不成立須等到資源足夠。

    - ∗ 暫時將 claim edge 改為 allocation edge，檢查是否有 cycle 存在（包含其他 claim edges），若無表示 safe；若有則 unsafe，並改回 claim edge。

Theorem (145, 148) Deadlock detection and recovery：

- 若所有類型資源皆為 single-instance，可用畫圖簡化 procedure。
  - 使用 Wait-for graph，省略 resources。
  - NO cycle, NO deadlock; 有 cycles，deadlock。
- Recovery：
  - Process termination：
    - ∗ Abort all deadlocked processes：Cost high，先前工作白費。
    - ∗ Abort one process at a time until deadlock is eliminated：每終止一個 deadlocked process，都要執行一次 deadlock detection alogrithm，cost high。
  - Resource preemption：
    - ∗ Selecting a victim：Cost high，注意 starvation。
    - ∗ Rolling back：Difficult, returning to safe state.

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。

- Busy waiting (spinlock)：若 process 平均可以在 < context switch 時間內離開 loop，則 spinlock 則有利。

- Busy waiting **無法**完全避免，在 `Entry section` 和 `Exit section` 仍是 busy waiting 實現，然而 disable interrupt 可以避免 busy-waiting，但 disable interrupt 不適合 multiprocessors。

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
```

Algorithm 2 P_j of Algorithm 1 (two processes solution).

```
1: function P_j
2:     repeat
3:         while turn ≠ j do
4:         end while
5:         Critical section.
6:         turn := i
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 3 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 4 P_j of Algorithm 2 (two processes solution).

```
1: function P_j
2:     repeat
3:         flag[j] := True
4:         while flag[i] do
5:         end while
6:         Critical section.
7:         flag[j] := 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，可以通過 hardware、OS support 或 high-level software APIs 提供的 synchronization tools。
  - 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 5 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
```

Algorithm 6 P_j of Peterson's solution (two processes solution).

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

2.1.2 Hardware support

Theorem () Memory barrier (fence)：

- Strongly ordered：對於 memory 修改 on 1 processor is immediately visible to all other processers。
- Weakly ordered：對於 memory 修改 on 1 processor may NOT be immediately visible to all other processers。
- System ensures that any L/S operations are completed before any subsequent L/S operations are performed.

Theorem (176) Test-and-Set：

- TEST-AND-SET：

Algorithm 7 Test-and-Set.

```
1: function TEST-AND-SET(bool *Target)                ▷ Atomic.
2:     bool rv := *Target
3:     *Target := True
4:     return rv
5: end function
```

- Algotihm 1（錯誤）：
  - 共享變數：

```
bool lock := False
```

Listing 4: 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 8 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 5: Shared variables of Algorithm 2 (TEST-AND-SET).

– 若移除 `waiting[i] := False`，則違反 progress，若僅 P_i 和 P_j 想進入 critical section，$waiting[i], waiting[j] = True$，且 P_i 先進入 critical section，$lock, waiting[i] := True$；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 $waiting[i] := False$ 而 $lock = True$，未來沒有 processes 可以再進入 critical section，deadlock。

Algorithm 9 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)**:**

- COMPARE-AND-SWAP**:**

Algorithm 10 Compare-and-Swap.

```
1: function COMPARE-AND-SWAP(bool *value, int expected, int new_value)   ▷ Atomic.
2:     int tmp := *value
3:     if *value = expected then
4:         *value := new_value
5:     end if
6:     return tmp
7: end function
```

- 共享變數**:**

```
int lock := 0
```

Listing 6: Shared variables of Algorithm 1 (COMPARE-AND-SWAP).

- 與 Test-and-Set Algorithm 1 (8) 類似，**Bounded waiting** 不成立。

- 正確 algorithm：參考 Test-and-Set Algorithm 2 (9) 將 `key := Test-and-Set(& lock)` 改為 `key := Compare-and-Swap(&lock, 0, 1)` 。

Algorithm 11 P_i (Compare-and-Swap).

```
1: function COMPARE-AND-SWAP
2:     repeat
3:         while COMPARE-AND-SWAP(&lock, 0, 1) ≠ 0 do
4:         end while
5:         Critical section.
6:         lock := 0
7:         Remainder section.
8:     until False
9: end function
```

- Atomic value：

Algorithm 12 P_i (atomic value).

```
1: function INCREMENT(atomic_int *v)
2:     tmp := *v
3:     while tmp ≠ COMPARE-AND-SWAP(v, tmp, tmp + 1) do
4:         tmp := *v
5:     end while
6: end function
```

2.1.3 Mutex lock

Theorem () Mutex lock：

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

```
bool Available := True
```

Listing 7: Shared variables of Mutex lock.

Algorithm 13 *acquire*().

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

Algorithm 14 *release*().

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

2.1.4 Semaphore

Theorem (178) Semaphore：

- OS software tools (system call).
- 為一種 integer data type。

Algorithm 15 *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 16 *signal*(S) ($V(S)$).

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

- 共享變數：

```
semaphore mutex := 1
```

Listing 8: Shared variables of semaphore.

Algorithm 17 P_i (semaphore).

```
1: function SEMAPHORE
2:     repeat
3:         WAIT(mutex)
4:         Critical section.
5:         SIGNAL(mutex)
6:         Remainder section.
7:     until False
8: end function
```

- semaphore 初值：1：互斥控制；0：強迫等待其他 process；n：允許 n 個 processes 同時運行。
- Liveness：Refers to a set of properties that a system must satisfy to ensure processes make progress, and indefinite waiting is an example of a liveness failure, e.g. deadlock, starvation.

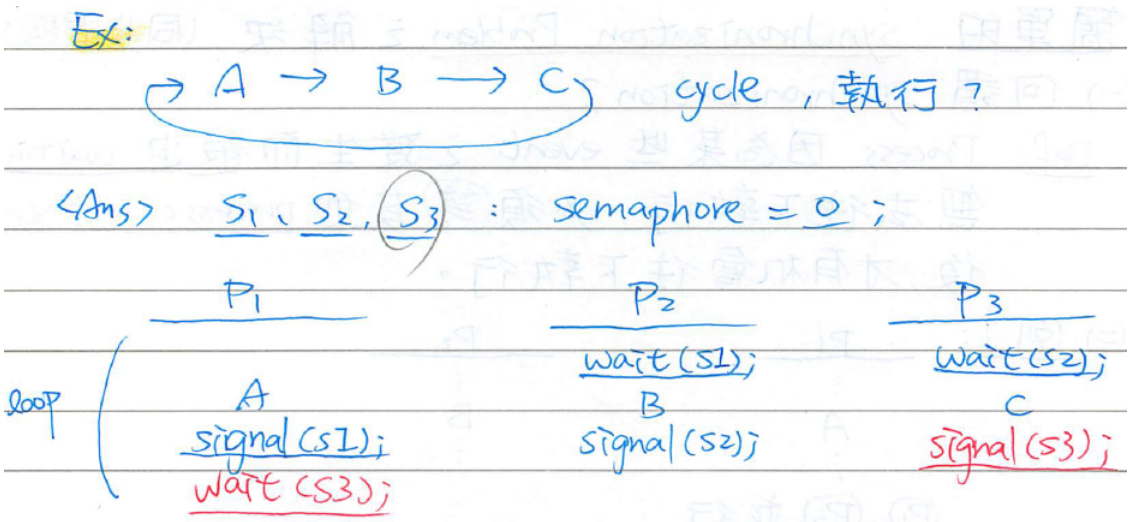

Figure 3: Example of semaphore.

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 18 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 19 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：

- Reader 和 writer 以及 writer 和 writer 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 20 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 21 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 22 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 23 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 24 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 25 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 26 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 (4)，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 27 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 waiting，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 s_1 := 1 // c互斥控制。
binary_semaphore s_2 := 0 // c < 0時卡住process
```

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

Algorithm 28 $wait(c)$ (counting semaphore).

```
1: function WAIT(c)
2:     WAIT(s_1)
3:     c := c − 1
4:     if c < 0 then
5:         SIGNAL(s_1)
6:         WAIT(s_2)                          ▷ Process 卡住。
7:     else
8:         SIGNAL(s_1)
9:     end if
10: end function
```

Algorithm 29 *signal*(c) (counting semaphore).

```
1: function SIGNAL(c)
2:     WAIT(s_1)
3:     c := c + 1
4:     if c ≤ 0 then                                  ▷ 先前有 process 卡住。
5:         SIGNAL(s_2)
6:     end if
7:     SIGNAL(s_1)
8: end function
```

Theorem () Non-busy waiting semaphore：

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

Listing 16: Non-busy waiting semaphore.

Algorithm 30 *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 31 *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 32 *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 33 *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 (9) 或 COMPARE-AND-SWAP (2.1.2) 實現。

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

Algorithm 34 *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:         Nop.
6:         Disable interrupt.
7:     end while
8:     S := S − 1
9:     Enable interrupt.
10: end function
```

Algorithm 35 *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.
- Definition：
  - Shared variables.
  - A set of functions.
  - Initialization code.
- 只有 monitor 內 functions 可以直接存取 shared memory。
- Monitor 本身確保 mutual exclusion，同一時間最多允許一個 process 呼叫 monitor 的任一 function。

- Programmer 不需煩惱 shared variables 的 race condition problem，只需解決 synchronization problem。
- 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 36 *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 37 *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 38 *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 39 *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 40 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 41 *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 42 *Release*().

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

Algorithm 43 *initialization_code*().

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

Algorithm 44 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 45 *Wait*().

```
1: function WAIT
2:     if value ≤ 0 then
3:         x.WAIT
4:     end if
5:     value := value − 1
6: end function
```

Algorithm 46 *Signal*().

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

Algorithm 47 *initialization_code*().

```
1: function INITIALIZATION_CODE
2:     value := 1
3: end function
```

Theorem () Monitor 種類：Process P call `signal` 將 process Q 恢復執行，則

- – Type 1：P waits Q until Q finished or Q is blocked again，多一個儲存這種 P 的 queue。又稱 Hoare monitor。
  - Type 2：Q waits P until P finished or P is blocked. 不保證 Q 可以 resume execution，因為 P 繼續執行很有可能改變 Q 可以恢復執行的條件。
  - Type 3：P 先離開，放到 entry queue 的第一位。保證 Q 一定可以恢復執行，但 NOT powerful than Hoare monitor，因為只能一進一出。
- – signal-and-wait：Type 1, Type 3.
  - signal-and-continue：Type 2.

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 functions body 前後加入控制碼，類似 Entry section 和 Exit section。

Algorithm 48 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 49 $x.wait()$.

```
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 50 $x.signal()$.

```
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 () 因為 monitor 和 semaphore 可以互相製作，所以解決 synchronization problem 能力相同。

Theorem (195) Indirect communication 通過 shared mailbox；溝通雙方可多條 communication links 且可與其他 processes 共享，direct 則否。

Theorem (197) Link capacity：

- 每一條 communication link 皆有一個 messages queue，而 queue size 即 link capacity。
- receiver：收到訊息才能往下執行；Sender：由 link capacity 決定 synchronization mode (zero (rendezvous，即雙方都在等), bounded, unbounded capacity)。

Theorem (223)

- Dynamic loading：不需要 OS 協助，programmer 責任。
- Dynamic linking：需要 OS 協助，因為 processes 間無法 access 彼此的 memory。

Theorem (229, 230, 238) Contiguous allocation：

- External fragmentation：
  - Compaction：
    - 不易制定 optimal policy。
    - process 必須是 dynamic binding 才行在 execution time 移動 memory blocks。
  - Page memory management：
    - 但有 internal fragmentation problem，page size 越大越嚴重。
    - Support memory sharing, memory protection, dynamic loading and dynamic linking.

    - ∗ Effective Memory Access Time (EMAT) 較久。
    - ∗
$$\begin{aligned}\text{EMAT} = p \times (\text{TLB time} + \text{Memory access time}) \\ + (1-p) \times (\text{TLB time} + 2 \times \text{Memory access time})\end{aligned} \tag{9}$$
      p is TLB hit rate.
- Internal fragmentation：
    - Segment memory management：
        - ∗ 但有 external fragmentation problem。
        - ∗ Support memory sharing and memory protection, 但比 paging 容易實施。
        - ∗ EMAT 更久，因為多了 checking $offset < limit$。
- Paged segment memory management：
    - 先 segment 再 page。
    - Segment table 紀錄 limit 和 page table address，且每個 segment 都有各自 page table。
    - 解決 external fragmentation problem，但有 internal fragmentation problem。
    - EMAT 最長，很佔空間。

Theorem (242) Virtual memory：

- OS 責任。
- Partial loading 即可執行，同時間可 load process 變多，multiprogramming degree 和 CPU utilization 皆上升，thrashing 除外。
- Less I/O time，但總體 I/O time 上升，因為次數提升。
- Memory utilization 皆上升，thrashing 除外。
- $$\text{EMAT} = (1-p) \times \text{Memory Access time} + p \times \text{Page fault time} \tag{10}$$
  p is page fault rate.

Theorem (246) Page replacement：

- 若 dirty bit 為 1，則多一次 swap out I/O，共兩次。

- Page replacement algorithms：
  - OPT：選擇將來長期不會使用的 page；最佳，但無法實現。
  - LRU：reverse OPT；page fault rate 可接受，但製作 cost 很高，需要大量 hardware support。
  - LRU 近似：
    - Additional reference bit usage：
      - 每一個 page 有一個 register 紀錄最近 n 次 reference bit 的值。
      - 每格一段時間將 register 右移一位，將 reference bit 作為最高位，並 reset reference bit。
      - Victim page 為 register 值最小者，若多個 pages 相同，則採用 FIFO。
    - 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。
  - Counting：若多個 pages 相同，則採用 FIFO；page fault rate 相當高；製作 cost 很高。LFU：選擇累計次數最少；MFU：選擇累計次數最多。
- 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 stack property。
- 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。
    - 與法一類似，free frames pool 的 free frames 同時記錄 process ID 及其 page no.，因為 free frames 內放的一定是最新的內容，因此可以先從 free frames pool 中找。

Theorem (253) Frame allocation：

- Process 可分配 frames 數量由 harware 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。
- Thrashing：
    - 當 process 的 frames 不足經常 page fault 且要 page replacement，若 OS 採用 global page replacement，則可能會造成其他 processes 也 page fault，因此幾乎所有 processes page fault，ready queue 為空，CPU utilization 下降，則 system 引入更多 processes 執行，造成情況惡化，同時 paging disk 異常忙碌。
    - 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。
    - 不容易制定精確 working set。
    - 若前後期 working set 內容差距太大，可能造成較多 I/O。
-
    - 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 (263) TLB reach：

- 通過 TLB 可以 access 的 size，希望越大越好。

$$\text{TLB reach} = \#\text{ of TLB entries} \times \text{page size} \tag{11}$$

- 增加 entry 數量：TLB reach 增加，但也有可能無法涵蓋 working set。

- 加大 page size：TLB reach 增加，page fault ratio 下降，但 internal framentation 較嚴重。解法：提供多種 page size，且在 TLB 增加 page size 欄位，因此許多 OS 改為管理 TLB。

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

Theorem (302) Disk free space management：

- Grouping：Linked list，同時記錄更多 free blocks 的 no.。
- Counting：Linked list，適用於 contiguous allocation and free，同時記錄接著的 contiguous free blocks。

Theorem (304) File allocation methods：

- Physical directory 紀錄。
- Contiguous allocation：
  - Seek time 較短，因為多數落在同一個 cylinder。
  - Support random (direct) access 和 sequential access 且後者較 linked allocation 快。
  - Reliablity 較 linked allocation 高。
  - External fragmentation problem，disk 中通過 repack 解決，類似 compaction。
  - 檔案大小不易擴增，需事先宣告大小。
- Linked allocation：
  - 優缺點與 contiguous allocation 相反。
  - 無法 support random access。
  - FAT (File Allocation Table)：將 linking info 存在 FAT (memory) 中，不存在 allocation disk，加速 access。
- Index allocation：
  - 不一定連續，需額外 allocate index block 記錄所有 allocated blocks no.。
  - Support random access 和 sequential access。
  - 可動態擴充，不事先宣告大小。
  - Linking info 比 linked allocation 大很多。

  - 若檔案大到一個 index block 無法紀錄。解決：
    - 多個 index blocks 用 linking 串連，I/O 次數大增。
    - Multilevel index structure：I/O 次數固定，但對小型檔案極不合適，index block 甚至比 data block 還多。
    - UNIX 的 I-Node：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：公平，no starvation，效果不好。
- SSTF (Shortest Seek-time Track First)：不公平，可能 starvation，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。
- C-SCAN (Circular SCAN)：只提供單向服務，回程不服務。
- Look：類似 SCAN，服務完該方向最後一個就折返。
- C-Look：只提供單向服務的 Look。
- 無最差也無最佳。

Theorem (313)

- Raw I/O：將 disk 視為大型，不支援 file system service，performance 佳，但 user 不方便使用。
- Bootstrap loader：只用在開機時將 OS 的 object code 從 disk 載入 RAM，放在 disk 中固定的 boot blocks。

Theorem (314) Bad sectors 處理：

- Spare sectors (sector sparing, forwarding)：
  - e.g. SCSI.
  - 在 low-level formatting 完成，OS 看不到。
  - 破壞 disk scheduling 效能。解法：把 spare sectors 分散到每個 cyclinder 上，bad sectors 利用鄰近的 spare sector。
- Sector slipping：移動其他 sectors，空出下一個 sector 取代 bad sector。

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.