作業系統
Operating System

TZU-CHUN HSU[1]

[1]vm3y3rmp40719@gmail.com

[1]Department of Computer Science, Zhejiang University

2020 年 10 月 30 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

2.1 System types

Theorem (7) Multiprogramming system：

- Multiprogramming degree：存在系統內執行的 process 個數。
- 執行方式：
  - Concurrent（並行）：CPU 在多個 processes 之間切換，同一時間只有一個 process 執行。
  - Parallel（平行）：同一時間多個 processes 在不同 CPU 上執行。

Theorem (7) Time-sharing (Multitasking) system：

- 屬於 Multiprogramming 的一種。
- CPU 頻繁在任務間切換，response time 短，適合 interactive computing，經常採用 Round-Robin scheduling。
- 採用 virtual memory。
- 採用 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 () Multiprocessors (Tightly-coupled, Parallel) system：

- Processors 溝通通常採用 shared memory。
- Graceful degradation (Fail-soft)：系統不會因為某些軟硬體元件故障而停頓。具備此能力的系統稱為 Fault-tolerant。
- Multicores，即一晶片多核，相較 multiprocessors 速度快且省電。
- 類型：
  - SMP (Symmetric MultiProcessors)：所有 processors 能力與權利相同，troughput 與 reliability 較 ASMP 好，但 OS 設計較 ASMP 複雜。
  - ASMP (ASymmetric MultiProcessors)：Master-slave 架構，master 負責工作與資源分配，其餘為 slave。

Theorem (8) Distributed (Loosely-coupled) system：

- Processors 的 clock 和 OS 未必相同。
- Processors 溝通通常採用 message passing，透過 network 連接。
- C/S model 達到 resources sharing；P2P (Peer-to-peer) 不區分 C/S。

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 共存，但是 real-time processes 不能被 swapped out 直到完工。
    - 可與 time-sharing 並存。
    - 現行 OS 皆支援。

Theorem () Batch system：非急迫或週期性的任務一起執行，提高資源使用率，不適用 real-time 及 user-interactive。

2.2 I/O 運作方式

Theorem (16) Polling (Busy-waiting, Programmed)：CPU 需不斷 polling I/O device controller 直到完成。

Theorem (16) Interrupted：

- I/O 完成時，I/O device controller 會 interrupt，使 CPU 暫停目前 process 放到 ready state queue。
- 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。
  -
    - Interrupt：Hardware-generated，例如 device controller 發出 I/O-completed。
    - 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，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)

- 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。

- I/O 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 (22) CPU protection：利用 timer，時間到 timer 發出 timer-out interrupt 通知 OS，讓 OS 強制取回 CPU。

2.3 Virtual machine

Theorem (34) Virtual machine：

- Host：Underlying hardware system.
- Virtual Machine Manager (VMM, Hypervisor).
- Guest：Process provided by VM, for example, OS, applications.
- 優點：
  - 提供 Cloud computing。
  - 測試中的 OS 掛了，視為一個 user process 掛掉，其他 user processes 以及 real system 不受影響。
- 缺點：
  - 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。

Theorem ()

- 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.

2.4 Process

Theorem (61) Process control block (PCB)：

- Process ID.
- Prcoess state.
- CPU register (stack top pointer, accumulator).
- Program Counter (PC).
- CPU scheduling info (PCB pointer, process priority).
- Memory management info (base/limit registers, page table).
- Accounting info (CPU time).
- I/O-status info (Uncompleted I/O request, allocated I/O devices).

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：Prcoess is allocated memory space, being put in ready queue. Ready for execution, being able to get CPU allocation.
  - Waiting (Block, Sleep in memory)：Waiting for some event to occur, being put in waiting queue, e.g. waiting for I/O-completed. Unable to get CPU allocation.
  - Terminated (Exit, Zombie)：Process is finished.
- Transition：
  - Admitted：Prcoess 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.
  - Exit：Process is finished, releasing resources.
  - Interrupt (Time-out)：Process release CPU, putting on ready queue, e.g. time-out interrupt.
  - I/O event wait：Process waits for I/O-completed or event occurs.
  - I/O-completed or event occurs：I/O-completed or event occurs, putting in ready queue.
  - Only in Stalling version (Figure 2)：
    * Swap out：Out of memory, and other process needs more space. Medium-term scheduler can choose some **Blocked** processes to swap out to disk with their process image being saved.
    * Swap in：Sufficient memory space. Medium-term scheduler can choose some Suspended/Ready processed to swap in from disk to memory, putting in ready state.
    * Suspend：Out of memory even all Blocked processes are swapped out or Blocked processes got higher priority than Ready processes. Swap out some **Ready** processes to get sufficient memory space.

  - (Poor design) Running to Suspended/Ready：If a higher priority Suspended/Block process becomes Ready, kernel can force a lower priority Running process to release CPU and memory space, putting the former in running queue.
  - (Poor design) Suspended/Block to Suspended/Ready：If as Suspended/Blocked got higher priority than a Running process, the latter release CPU and memory space, swapping in the former and get CPU allocation.

- 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.

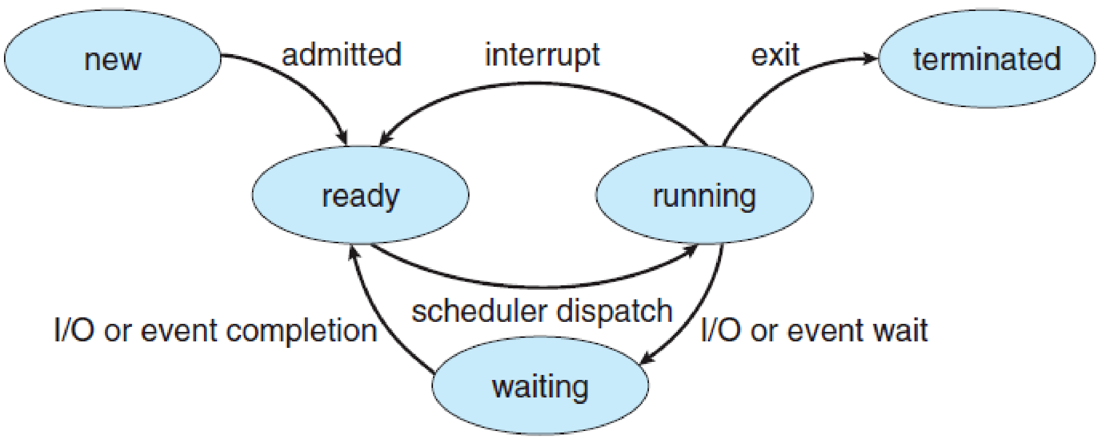

Figure 1: Process life cycles.

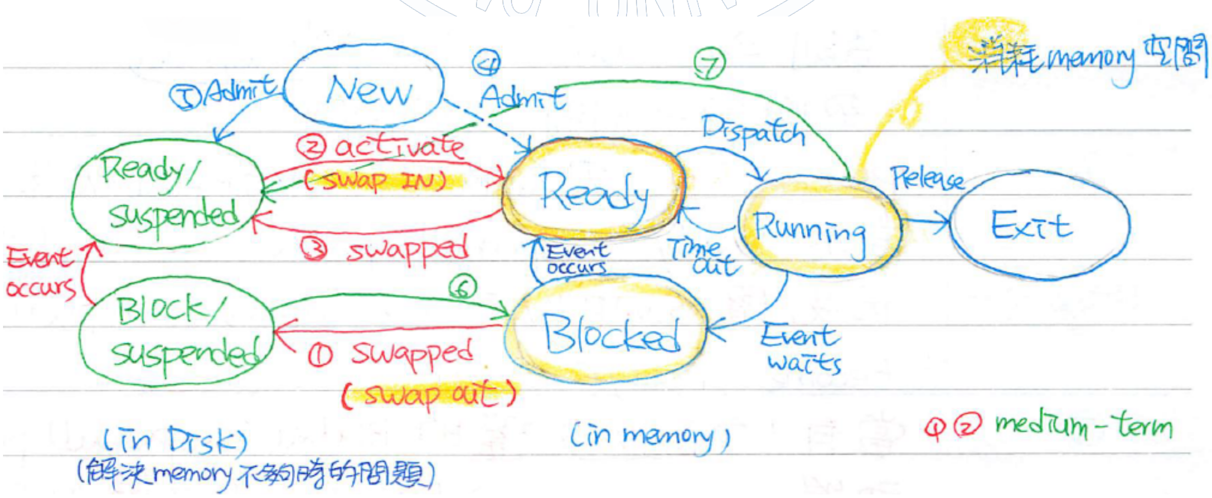

Figure 2: Process life cycles (Stalling version).

2.5 Scheduling

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：

- CPU utilization：$\frac{\text{CPU process execution time}}{\text{CPU total time}}$。
- Throughput：單位時間完成 process 數量。
- 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：
  - Process 長期無法取得完成工作所需資源，因此遲遲無法完成，形成 indefinite blocking。
  - 容易發生在不公平或是 preemptive 的 scheduling。
  - 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。
- Non-preemptive (Cooperative)：
  - 完成工作時間較可預期。
  - Context switch 次數較少。
  - 較少發生 Race condition，特別是 kernel。
  - Scheduling performance 較差，平均等待時間較長。

- Real-time 和 time-sharing systems 不適合。

- Preemptive：
  - 大部分 OS 在 kernel mode 採用，防止 race condition。
  - 對 soft real-time systems 不利，例如 kernel 執行時

2.5.1 Scheduling algorithms

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。

- Priority scheduling：FCFS, SJF, SRTF $\subset$ Priority scheduling
- Round-Robin (RR) scheduling：
  - Time-sharing system 採用。
  - FCFS $\subset$ RR
  - 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：
  - Queue 之間採用 fixed priority preemptive scheduling 或 RR。
  - 每個 queue 有各自的 scheduling。
  - process 不能在 queues 之間移動，因此缺乏彈性。
  - 易 starvation，且無法通過類似 Aging 改善。
- Multi-level feedback queue：
  - 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。
  - 所有 scheduling superset。
- Combination of RR and Priority scheduling：多個 processes 有相同 priority 時，採用 RR。

Scheduling	公平	Preemptive	Non-preemptive	Starvation
FCFS	√		√	
SJF			√	√
SJF		√		√
Priority		√	√	√
RR	√	√		
Multi-level queue		√		√
Multi-level feedback queue		√		

2.5.2 特殊系統 scheduling

Theorem (79) Multiprocessors system：

- 包含多 CPUs，Multicores，NUMA system，Heterogeneous（手機上多核 CPU）system，Multithreaded（硬體）system。
- 多 CPUs：
  - ASMP：類似 single-CPU scheduling。
  - SMP：分 2 approaches：
    - 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， it cost 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 3)，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。

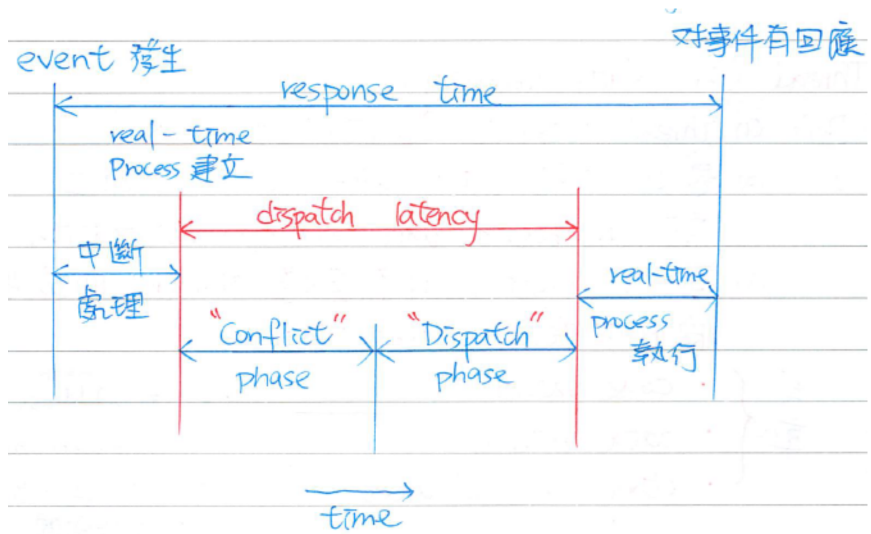

Figure 3: Dispatch latency on real-time systems.

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：

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

  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：
    * 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%。

2.6 Thread

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。
- 每一條 thread 有 private Thread Content Block (TCB)，包含 PC、registers、stack 和 local variables 等，因此須防止 race condition。
- 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.

2.7 Deadlock

Theorem (136) Deadlock：

- **必要**條件：
  - Mutual exclusion：同一個時間點資源只允許一個 process 持有。
  - Hold and wait：持有部分資源又在等待其他 process 持有的資源。

  - No preemption：process 不可任意搶奪其他 process 持有的資源，必須等他自願釋放，才有機會取得。
  - Circular wait：多個 processes 形成循環等待的關係。

- deadlock throughput 低，但 starvation 與效能差無必然關係，例如 SRTF 有 starvation，但 scheduling 效益最佳。
- 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：

- 保證 no deadlock。
- Utilization 和 throughput 較低。
- 可能 starvation。
- 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，需先放掉所持有的資源。
  - No preemption：改為 preemption。
    * 須防止 race condition。
    * Resource recovery.
    * 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：

- 保證 no deadlock。

- Utilization 和 throughput 較低。
- 可能 starvation。
- Banker's algorithm：
  - n processes, m resources.
  - $Request_i$: P_i 提出的資源申請量。
  - Max：processes 完成工作所需最大的各種資源數量。
  - $Allocation$：processes 所持有的各種資源數量。
  - $Need$：processes 所缺少的各種資源數量。
  - $Available$：目前可用的各種資源數量。
  - Procedures：
    * Check $Request_i \leq Need_i$，若不成立直接終止。
    * Check $Request_i \leq Available$，若不成立須等到資源足夠。
    * 暫時更新狀態，執行 safety algorithm。
  - Safety algorithm：
    * $Work$：即 $Available$。
    * $Finish$：表示是否可完工，初始值為 $False$。
    * Procedures：Recursively check 所有未完工的 processes，若滿足 $Finish_i = False \land Need_i \leq Work$，則 $Finish_i = True$; $Work$ += $Allocation_i$；否則若 $Finish$ 皆為 $True$，返回 $safe$，反之返回 $unsafe$。
  - Time complexity: $O(n^2 \times m)$.
- 若 n processes，m resources（單一種類），若滿足

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

則保證不會 deadlock。證明：若所有資源都分配給 processes，即 $\sum_{i=1}^{n} Allocation_i = m$，因為 $\sum_{i=1}^{n} Need_i = \sum_{i=1}^{n} Max_i - \sum_{i=1}^{n} Allocation_i \rightarrow \sum_{i=1}^{n} Max_i = \sum_{i=1}^{n} Need_i + m$，根據第二條件，$\sum_{i=1}^{n} Max_i < n + m$，有 $\sum_{i=1}^{n} Need_i < n$，表示至少一個 process P_i，$Need_i = 0$，因為 $Max_i \geq 1 \land Need_i = 0 \rightarrow Allocation_i \geq 1$，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，若無表示 safe；若有則 unsafe，並改回 claim edge。
- Unsafe 未必 deadlock。

Theorem (145, 148) Deadlock detection and recovery：

- 不保證 no deadlock。
- Utilization 和 throughput 較高。
- 偵測系統**目前**是否 deadlock，若有則 recovery。
- Deadlock detection algorithm：
  - n processes，m resources.
  - $Allocation, Request, Available, Work, Finish$
  - Procedures：
    * 設定初始值：

$$Work = Available$$
$$Finish_i = \begin{cases} False & , Allocation_i \neq 0 \\ True & , Allocation_i = 0 \end{cases} \quad (4)$$

    * Recursively check $Finish_i = False \wedge Request_i \leq Work$，則 $Finish_i = True; Work \mathrel{+}= Allocation_i$；否則 check $Finish$ 皆為 $True$，則 no deadlock，否則 $Finish_i = False$，P_i deadlock。
  - Detection 一次 time complexity $O(n^2 \times m)$，另位還要考慮偵測頻率，因此 cost 極高。

- 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 高，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：Minimizing cost，cost 高，注意 starvation。
    - Rolling back：Difficult, returning to safe state.

Theorem () Process communication：

- Shared memory：
  - OS 不須提供額外支援，just provides shared memory。
  - 須防止 race condition。
  - 適用大量資訊溝通，且通訊速度快，因為不須 kernel 支持。
  - 較不適合分散式系統。
- Message passing：特點與 shared memory 相反。

Theorem () Kernel 也會有 race condition problem。

- Non-preemptive kernel：Kernel 在執行 system processes 時，不會被其他 process preempted，所以不會有 race condition problem，但是對 user process response 較差。
- Preemptive kernel：More responsive and more suitable for real-time programming，但可能會有 race condition。

Theorem () Disable interrupt：

- 執行 shared variables 存取前先 disable interrupt，完成後再 enable interrupt，期間 CPU 不會被 preempted。

- Simple，適用於 uniprocessor (single-core)。
- 不適合用於 multiprocessors (multi-cores)，因為只能 disable all CPUs interrupt，導致 parallel performance 很差。
- 風險很高，因為 user process 可能不 enable interrupt。

2.8 Critical section design

Theorem (170) Introduction to critical section：

- 設計 entry section 和 exit section 的控制程式。
- 在 critical section，CPU 也可能 preempted。
- 適用 multiprocessors。
- 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) 技巧：
  - 在 entry section 設計中，常用來迫使 process waiting，無法往下執行，即利用 loop，e.g. `for`，`while`。
  - 若平均 waiting time 很長，則 spinlock 不利，因為 waiting process 仍要和其他 process 爭奪 CPU，且將拿到的 CPU time 用在無進度的 loop 上。
  - 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。

2.8.1 Software support

Theorem (171) Two processes solution：

- Algorithm 1：
  - 共享變數：$int\ turn$ 初始值為 i 或 j。
  - Mutual exclusion：成立，不會同時進入。
  - 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。
  - Bounded waiting：成立。

Algorithm 1 P_i of Algorithm 1 (two process solution).

function P_i
    repeat
        while $turn \neq i$ **do**
        end while
        Critical section.
        $turn := j$
        Remainder section.
    until $False$
end function

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

function P_j
    repeat
        while $turn \neq j$ **do**
        end while
        Critical section.
        $turn := i$
        Remainder section.
    until $False$
end function

- Algoroithm 2：
  - 共享變數：$boolean\ flag$ 初始值為 $False$，表示是否想進入 critical section。
  - Mutual exclusion：成立，不會同時進入。
  - 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。

- Bounded waiting：成立。

Algorithm 3 P_i of Algorithm 2 (two process solution).

function P_i
    repeat
        $flag[i] := True$
        while $flag[j]$ **do**
        end while
        Critical section.
        $flag[i] := False$
        Remainder section.
    until $False$
end function

Algorithm 4 P_j of Algorithm 2 (two process solution).

function P_j
    repeat
        $flag[j] := True$
        while $flag[i]$ **do**
        end while
        Critical section.
        $flag[j] := False$
        Remainder section.
    until $False$
end function

- Algorithm 3 (Peterson's solution):
  - 共享變數：
    - $int\ turn$ 初始值為 i 或 j。
    - $boolean\ flag$ 初始值為 $False$，表示是否想進入 critical section。
  - Mutual exclusion：成立，若雙方都想進入 critical section，且當執行到 `while`，$turn$ 都已經設定給對方，則較早執行者可以進入 critical section。
  - Progress：成立。
  - Bounded waiting：成立。
  - $flag$ 或 $turn$ 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 process solution).

function P_i
    repeat
        $flag[i] := True$
        $turn := j$
        while $flag[j] \wedge turn = j$ **do**
        end while
        Critacal section.
        $flag[i] := False$
        Remainder section.
    until $False$
end function

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

function P_j
    repeat
        $flag[j] := True$
        $turn := i$
        while $flag[i] \wedge turn = i$ **do**
        end while
        Critacal section.
        $flag[j] := False$
        Remainder section.
    until $False$
end function

2.8.2 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：

- 回傳 $Target$ 的舊值且將 $Target$ 一律設為 $True$ 且為 atomically executed。

Algorithm 7 Test-and-Set.

function TEST-AND-SET(boolean $*Target$)
    boolean $rv := *Target$
    $*Target := True$
    return rv
end function

- Algotihm 1：
  - 共享變數：*boolean lock* 初始值為 $False$。
  - Mutual exclusion：成立。
  - Progress：成立。
  - 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).

function P_i
    repeat
        while TEST-AND-SET($\&lock$) **do**
        end while
        Critical section.
        $lock := False$
        Remainder section.
    until $False$
end function

- Algotihm 2：
  - 共享變數：
    - *boolean lock* 初始值為 $False$。
    - *boolean waiting*$[0 \cdots n-1]$ 初始值為 $False$。

$$waiting[i] = \begin{cases} True & \text{,表示}P_i\text{想進入 critical section 且正在等待中} \\ False & \text{,不用等待，}P_i\text{可以進入或是} \\ & \text{已經在 critical section 中（初值無意義）} \end{cases} \tag{5}$$

- Mutual exclusion：成立，第一個執行 TEST-AND-SET 的 process，才能將 key 設為 $False$ 離開 `while`；或是依靠離開 critical section 的 process 修改唯一一個 process 的 $waiting$ 為 $False$。
- Progress：成立。
- Bounded waiting：成立。
- 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).

```
function P_i
    repeat
        waiting[i] := True
        key := True                                   ▷ Local variable.
        while waiting[i] ∧ key do
            key := TEST-AND-SET(&lock)
        end while
        waiting[i] := False
        Critical section.
        j := i + 1  mod n
        while j ≠ i ∧ ¬waiting[j] do                  ▷ 找下一個想進入的 P_j。
            j := j + 1 (mod n)
        end while
        if j = i then                                 ▷ 沒有 P_j 想進入 critical section。
            lock := False
        else
            waiting[j] := False
        end if
        Remainder section.
    until False
end function
```

Theorem (177) Compare-and-Swap (CAS)：

- 若 $*value = expected$，則 $*value := new\_value$，且一律回傳 $*value$ 舊值，同時保證 atomically executed。

Algorithm 10 Compare-and-Swap.

```
function Compare-and-Swap(boolean *value, int expected, int new_value)
    int tmp := *value
    if *value = expected then
        *value := new_value
    end if
    return tmp
end function
```

- 共享變數：*int lock* 初始值為 0。
- 與 Test-and-Set Algorithm 1 (8) 類似，Mutual exclusion 和 Progress 成立，而 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).

```
function Compare-and-Swap
    repeat
        while Compare-and-Swap(&lock, 0, 1) ≠ 0 do
        end while
        Critical section.
        lock := 0
        Remainder section.
    until False
end function
```

- Atomic value：

Algorithm 12 P_i (atomic value).

```
function Increment(atomic_int *v)
    tmp := *v
    while tmp ≠ Compare-and-Swap(v, tmp, tmp + 1) do
        tmp := *v
    end while
end function
```

2.8.3 Mutex lock

Theorem () Mutex lock：

- OS software tools (system call).
- Functions：共享變數：$boolean\ Available$ 初始值為 $True$。

Algorithm 13 $acquire()$.

```
function ACQUIRE
    while ¬Available do
    end while
    Available := False
end function
```

Algorithm 14 $release()$.

```
function RELEASE
    Available := True
end function
```

2.8.4 Semaphore

Theorem (178) Semaphore：

- OS software tools (system call).
- 為一種 integer data type。
- **Atomic** functions：

Algorithm 15 $wait(S)$ $(P(S))$.

```
function WAIT(S)
    while S ≤ 0 do
    end while
    S := S − 1
end function
```

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

```
function SIGNAL(S)
    S := S + 1
end function
```

- 共享變數：$semaphore\ mutex$ 初始值為 1。

Algorithm 17 P_i (semaphore).

```
function SEMAPHORE
    repeat
        WAIT(mutex)
        Critical section.
        SIGNAL(mutex)
        Remainder section.
    until False
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.

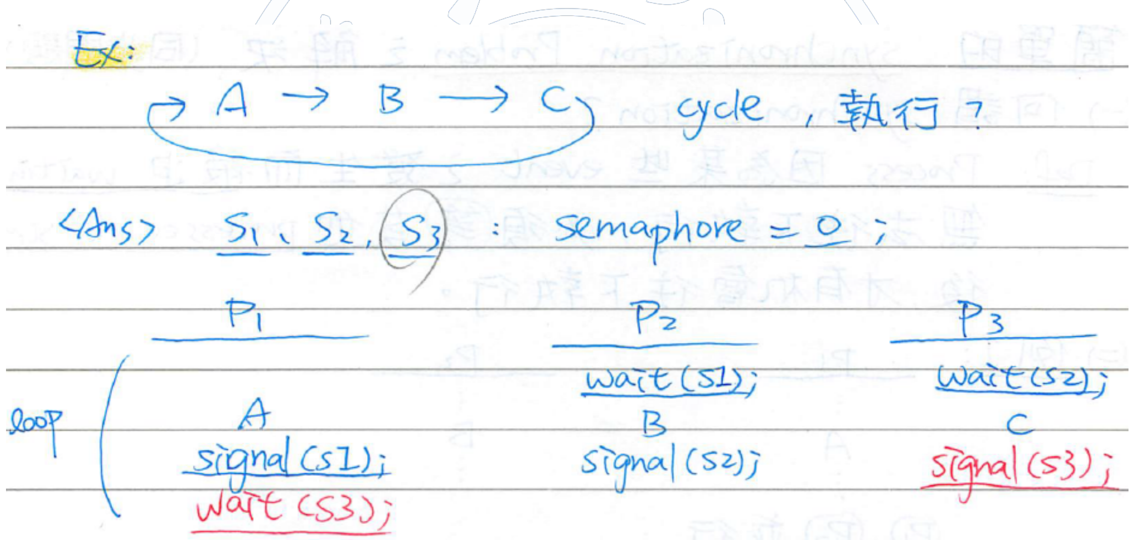

Figure 4: Example of semaphore.

Theorem (168) Producer-consumer problem：

- Producer 和 consumer 共用一個 buffer。討論 bounded buffer：buffer 大小有限。當 buffer 滿，producer 被迫等待；當 buffer 空，consumer 被迫等待。
- 共享變數：
  - *semaphore mutex* 初始值為 1。
  - *semaphore empty* 初始值為 n 表示 buffer 空格數。

  - *semaphore full* 初始值為 0 表示 buffer 中 item 數。

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

Algorithm 18 Producer.

```
function PRODUCER
    repeat
        Produce an item.
        WAIT(empty)
        WAIT(mutex)
        Add the item to buffer.
        SIGNAL(mutex)
        SIGNAL(full)
    until False
end function
```

Algorithm 19 Consumer.

```
function CONSUMER
    repeat
        WAIT(full)
        WAIT(mutex)
        Retrieve an item from buffer.
        SIGNAL(mutex)
        SIGNAL(empty)
        Consume the item.
    until False
end function
```

Theorem (182) Reader/Writer problem：

- Reader 和 writer 以及 writer 和 writer 皆要互斥。

- First readers/writers problem：

  - 對 readers 有利，writer 不利，writer 可能 starvation，即若已有 reader 在 read，即使後來有 writer 先來，後面接著 readers，writer 也必須等到所有 readers read 完才能 write。

  - 共享變數：

    * *semaphore wrt* 初始值為 1 作為 R/W 和 W/W 互斥控制同時用作對 writer 不利之阻擋。

    * *int readcnt* 初始值為 0 表示 reader 個數。

* *semaphore mutex* 初始值為 1 用作 *readcnt* 互斥控制。

Algorithm 20 Writer (First readers/writers problem).

```
function WRITER
    repeat
        WAIT(wrt)
        Writing.
        SIGNAL(wrt)
    until False
end function
```

Algorithm 21 Reader (First readers/writers problem).

```
function READER
    repeat
        WAIT(mutex)
        readcnt := readcnt + 1
        if readcnt = 1 then                ▷ 表示第一個 reader 需偵測有無 writer 在。
            WAIT(wrt)
        end if
        SIGNAL(mutex)
        Reading.
        WAIT(mutex)
        readcnt := readcnt − 1
        if readcnt = 0 then                ▷ No reader.
            SIGNAL(wrt)
        end if
        SIGNAL(mutex)
    until False
end function
```

- Second readers/writers problem：
  - 與 First readers/writers problem 相反，即若已有 writer 在 write，即使後來有 reader 先來，後面接著 writers，reader 也必須等到所有 wrtier write 完才能 read。
  - 共享變數：
    * *int readcnt* 初始值為 0 表示 reader 個數。
    * *semaphore mutex* 初始值為 1 用作 *readcnt* 互斥控制。
    * *semaphore wrt* 初始值為 1 作為 R/W 和 W/W 互斥控制。
    * *int wrtcnt* 初始值為 0 表示 writer 個數。

* *semaphore y* 初始值為 1 用作 *wrtcnt* 互斥控制。
* *semaphore rsem* 初始值為 1 用作對 reader 不利之阻擋。
* *semaphore z* 初始值為 1 為 reader 的入口控制，delay readers' speed，可有可無。

Algorithm 22 Writer (Second readers/writers problem).

```
function WRITER
    repeat
        WAIT(y)
        wrtcnt := wrtcnt + 1
        if wrtcnt = 1 then                ▷ 表示第一個 writer 需阻擋 readers。
            WAIT(rsem)
        end if
        SIGNAL(y)
        WAIT(wrt)
        Writing.
        WAIT(y)
        wrtcnt := wrtcnt − 1
        if wrtcnt = 0 then
            SIGNAL(rsem)                  ▷ No writer.
        end if
        SIGNAL(wrt)
        SIGNAL(y)
    until False
end function
```

Algorithm 23 Reader (Second readers/writers problem).

```
function READER
    repeat
        WAIT(z)
        WAIT(rsem)
        WAIT(mutex)
        readcnt := readcnt + 1
        if readcnt = 1 then
            WAIT(wrt)
        end if
        SIGNAL(mutex)
        SIGNAL(rsem)
        SIGNAL(z)
        Reading.
        WAIT(mutex)
        readcnt := readcnt − 1
        if readcnt = 0 then
            SIGNAL(wrt)
        end if
        SIGNAL(mutex)
    until False
end function
```

Theorem (184) The sleeping barber problem：

- 共享變數：
  - *semaphore customer* 初始值為 0 用作強迫 barber sleep。
  - *semaphore barber* 初始值為 0 用作強迫 customer sleep if barber is busy。
  - *int waiting* 初始值為 0 表示正在等待的客人個數。
  - *semaphore mutex* 初始值為 1 用作 *waiting* 互斥控制。
- 若將 BARBER 將兩行 `wait` 對調，可能會 deadlock。

Algorithm 24 Barber.

```
function BARBER
    repeat
        WAIT(customer)                    ▷ Barber go to sleep if no customer.
        WAIT(mutex)
        waiting := waiting − 1
        SIGNAL(barber)                    ▷ Wake up customer.
        SIGNAL(mutex)
        Cutting hair.
    until False
end function
```

Algorithm 25 Customer.

```
function CUSTOMER
    repeat
        WAIT(mutex)
        if waiting < n then               ▷ 入店。
            waiting := waiting + 1
            SIGNAL(mutex)
            SIGNAL(customer)              ▷ Wake up barber.
            WAIT(barber)                  ▷ Customer go to sleep if barber is busy.
            Getting cut.
        else
            SIGNAL(mutex)
        end if
    until False
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：
  - 共享變數：*semaphore chopstick*$[0\cdots 4]$ 初始值皆為 1 用作筷子互斥。
  - 會發生 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).

```
function P_i
    repeat
        Hungry.
        WAIT(chopstick[i])
        WAIT(chopstick[(i + 1 (mod 5))])
        Eating.
        SIGNAL(chopstick[i])
        SIGNAL(chopstick[(i + 1 (mod 5))])
        Thinking.
    until False
end function
```

- Algorithm 2：
  - 根據公式3，人數必須 < 5 才不會 deadlock。
  - 相較 Algorithm 1 (26)，額外添加共享變數：*semaphore no* 初始值皆為 4 用作可拿筷子的哲學家數量互斥。

Algorithm 27 Algorithm 2 P_i (The dining-philosophers problem).

```
function P_i
    repeat
        WAIT(no)
        (Same as Algorithm 1)
        SIGNAL(no)
    until False
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，偶數的哲學家先取左邊，再取右邊，奇數則反之，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 表示 counting semaphore 號誌值，e.g. 1。
  - *semaphore* s_1 初始值為 1 用作 c 互斥控制。

- *binary_semaphore* s_2 初始值為 0，當 $c < 0$ 卡住 process。

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

```
function WAIT(c)
    WAIT(s1)
    c := c − 1
    if c < 0 then
        SIGNAL(s1)
        WAIT(s2)                                   ▷ Process 卡住。
    else
        SIGNAL(s1)
    end if
end function
```

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

```
function SIGNAL(c)
    WAIT(s1)
    c := c + 1
    if c ≤ 0 then                                  ▷ 先前有 process 卡住。
        SIGNAL(s2)
    end if
    SIGNAL(s1)
end function
```

Theorem () Non-busy waiting semaphore：

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

Listing 1: Non-busy waiting semaphore.

Algorithm 30 *wait*(S) (non-busy waiting semaphore).

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

Algorithm 31 *signal*(S) (non-busy waiting semaphore).

function SIGNAL(S)
  $S.value := S.value + 1$
  if $S.value \leq 0$ **then**
    Remove process p from $S.Q$.
    $wakeup(p)$ ▷ System call 將 p 的 state 從 wait 改為 ready，有 context switch cost。
  end if
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).

function WAIT(S)
  Disable interrupt.
  $S.value := S.value - 1$
  if $S.value < 0$ **then**
    Add process p into $S.Q$.
    Enable interrupt.
    $block(p)$
  else
    Disable interrupt.
  end if
end function

Algorithm 33 *signal*(S) of Algorithm 1 (disable interrupt and non-busy waiting).

function SIGNAL(S)
  Disable interrupt.
  $S.value := S.value + 1$
  if $S.value \leq 0$ **then**
    Remove process p from $S.Q$.
    $wakeup(p)$ ▷ System call 將 p 的 state 從 wait 改為 ready，有 context switch cost。
  end if
  Enable interrupt.
end function

- Algorithm 2 (critacal section design and non-busy waiting)：將 Algorithm 1 (2.8.4) 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 (2.8.2) 實現。

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

Algorithm 34 *wait*(S) of Algorithm 3 (disable interrupt design and busy waiting).

function WAIT(S)
    Disable interrupt.
    while $S \leq 0$ **do**
        Enable interrupt.
        Nop.
        Disable interrupt.
    end while
    $S := S - 1$
    Enable interrupt.
end function

Algorithm 35 *signal*(S) of Algorithm 3 (disable interrupt design and busy waiting).

function SIGNAL(S)
    Disable interrupt.
    $S := S + 1$
    Enable interrupt.
end function

- Algorithm 4 (critical section design and busy waiting)：同 Algorithm 2 (2.8.4)，將 Algorithm 3 (2.8.4) 中的 `Enable interrupt.` 和 `Disable interrupt.` 分別改為 `Entry section.` 和 `Exit section.` 。

2.8.5 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 2: Data structure (The dining philosophers problem (Monitor)).

Algorithm 36 *pickup*(i).

function PICKUP(i)
    $state[i] := hungry$
    TEST(i)
    if $state[i] \neq eating$ **then**
        $self[i]$.WAIT
    end if
end function

Algorithm 37 *test*(i).

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

Algorithm 38 *putdown*(i).

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

Algorithm 39 *initialization_code*(i).

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

Algorithm 40 P_i (The dining philosophers problem (Monitor)).

function P_i
    Dining_ph dp ▷ Shared variable.
    repeat
        Hungry. ▷ No active.
        dp.PICKUP(i) ▷ Running: active; Blocked: NOT active.
        Eating. ▷ No active.
        dp.PUTDOWN(i) ▷ Active.
        Thinking. ▷ No active.
    until $False$
end function

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

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

```
}
```

Listing 3: Data structure (example (Monitor))

Algorithm 41 *Apply*(i).

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

Algorithm 42 *Release*().

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

Algorithm 43 *initialization_code*().

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

Algorithm 44 P_i of example (Monitor).

```
function P_i
    pa := PRINTERALLOCATION                ▷ Shared variable.
    n := pa.APPLY(i)
    Using printer pr[n] .
    pa.RELEASE(n)
end function
```

Theorem () 使用 Monitor 製作 binary semaphore：

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

```
        Condition x;
}
```

Listing 4: Data structure (Making semaphore using monitor).

Algorithm 45 $Wait()$.

---

function WAIT
    if $value \leq 0$ **then**
        x.WAIT
    end if
    $value := value - 1$
end function

Algorithm 46 $Signal()$.

---

function SIGNAL
    $value := value + 1$
    x.SIGNAL
end function

Algorithm 47 $initialization\_code()$.

---

function INITIALIZATION_CODE
    $value := 1$
end function

Theorem () Monitor 種類：process P call `siganl` 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：

- $semaphore\ mutex$ 初始值為 1 用作互斥控制。
- $semaphore\ next$ 初始值為 0 用來 block process P if P call `signal` 。
- $int\ next\_cnt$ 初始值為 0 用來統計 P 那種特殊 waiting processes 的個數。
- $semaphore\ x\_sem$ 初始值為 0 用來 block process Q if Q call `wait` 。
- $int\ x\_cnt$ 初始值為 0 用來統計一般 waiting processes 的個數。

- 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).

function F
 WAIT($mutex$)
 Function body.
 if $next\_cnt > 0$ **then**
  SIGNAL($next$)
 else
  SIGNAL($mutex$)
 end if
end function

Algorithm 49 $x.wait()$.

function x.WAIT
 $x\_cnt := x\_cnt + 1$
 if $next\_cnt > 0$ **then**
  SIGNAL($next$)
 else
  SIGNAL($mutex$)
 end if
 WAIT(x_sem)   ▷ Q 自己卡住。
 $x\_cnt := x\_cnt - 1$ ▷ Q 被救。
end function

Algorithm 50 $x.signal()$.

function x.SIGNAL
 if $x\_cnt > 0$ **then**
  $next\_cnt := next\_cnt + 1$
  SIGNAL(x_sem)
  WAIT($next$)   ▷ P 自己卡住。
  $next\_cnt := next\_cnt - 1$ ▷ P 被救。
 end if
end function

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

2.8.6 Message passing

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)。

2.9 Memory management

Theorem (222) Binding：

- 決定在 process 執行在 memory 起始 address。
- 時間點：
  - Compiling time by compiler. 通常只有 OS commands 採用，e.g. COM file。
  - Loading/Linking time by linking loader.
  - Execution time by OS, so called dynamic binding. MMU (Memory Management Unit)：Translation from logical (virtual) to physical address.

Theorem (223) Dynamic loading：

- Load-on-call，execution time 需要才 load。
- Process execution time 較久，且牽扯到 I/O。
- 不需要 OS 協助，programmer 責任。

Theorem (223) Dynamic linking：

- Modules code 在 execution time 需要才 link。
- 節省 object code space。

- Shared library，助於 library 更新。
- 需要 OS 協助，因為 processes 間無法 access 彼此的 memory。

Theorem (229, 230) Contiguous allocation：

- 被占用的 space 也稱 partition，通常是 variable，也稱 variable partition。Partititon 數量也是 process 數量，即 multiprogramming degree。
- Process 利用 linked list 管理 free memory blocks (holes)，稱作 Available list (AV-list)。
- External fragmentation：
  - Free memory blocks 總和夠大，但各自不夠。
  - Compaction：移動執行中的 process，使不連續 memory blocks 變連續。不易制定 optimal policy，且 process 必須是 dynamic binding 才行在 execution time 移動 memory blocks。
  - Page memory management：
- Internal fragmentation：配給 process 的 space 過多。

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. Tkb 筆記. https://drive.google.com/file/d/0B8-2o6L73Q2VelFZaXpBVGx2aWM/view?usp=sharing, 2017.