作業系統

Operating System

TZU-CHUN HSU[1]

[1]vm3y3rmp40719@gmail.com

[1]Department of Computer Science, Zhejiang University

2022 年 3 月 12 日

Version 5.0.0

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

2 Summary

Theorem (7, 10)

- Time-sharing (Multitasking)：使用 virtual memory 以及 spooling，且對所有 users 公平對待。
- Real-time：
  - **Hard** real-time disk 少用，**不**使用 virtual memory；但 **soft** real-time 可，但 real-time processes 的 pages 在完成前不能被 swapped out。
  - **Hard** real-time **不**與 time-sharing 並存；但 **soft** real-time 可。
  - 減少 kernel 干預時間，因為 Linux kernel 在執行某些 system process 時，**不**允許 user process preempts kernel，防止 race condition。

Theorem (16)

- Interrupt：Hardward-generated, e.g. I/O-complete, Time-out.
- Trap：Software-generated。Catch arithmetic error 或重大 error，例如 Divide-by-zero，以及 process 需要 OS 提供服務，會先發 trap 通知 OS。

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 switching.
- Switch mode to user mode.
- Jump to execution entry of user process.

Theorem (72, 78, 82, 84) CPU scheduling：

- Non-preemptive SJF 不適合用在 **short-term** scheduler，因為很難在短時間算出 next CPU burst；long-term scheduler 較合適。
- MFQ 雖然不公平，但 **NO** starvation。
- Linux 指定 processes 不要移轉到某些 processors。
- Worst-case CPU utilization for scheduling n processes using Rate-monotonic:

$$\begin{aligned} &2 \times (2^{\frac{1}{n}} - 1) \\ &\Rightarrow (n \to \infty) = 69\% \end{aligned} \tag{1}$$

- Dispatch latency：
  - Conflict phase：preempts kernel，並且 low-priority process releases needed resources for high-priority process。
  - Dispatch phase：Context switching, change mode to user mode, and jump to the user process.

Theorem (141) Deadlock avoidance：

- 若 n processes，m resources（單一種類），若滿足

$$\begin{aligned} 1 \leq Max_i \leq m \\ \sum_{i=1}^{n} Max_i < n + m \end{aligned} \tag{2}$$

則 NO deadlock。

Proof. 若所有資源都分配給 processes，即

$$\sum_{i=1}^{n} Allocation_i = m \tag{3}$$

又

$$\begin{aligned}\sum_{i=1}^{n} Need_i = \sum_{i=1}^{n} Max_i - \sum_{i=1}^{n} Allocation_i \\ \to \sum_{i=1}^{n} Max_i = \sum_{i=1}^{n} Need_i + m\end{aligned} \tag{4}$$

根據第二條件，有

$$\begin{aligned}\sum_{i=1}^{n} Max_i < n + m \\ \to \sum_{i=1}^{n} Need_i < n\end{aligned} \tag{5}$$

$\exists$ process P_i，$Need_i = 0$，又

$$\begin{aligned}Max_i \geq 1 \land Need_i = 0 \\ \to Allocation_i \geq 1\end{aligned} \tag{6}$$

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。

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 (Peterson's solution)：

- 共享變數：

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

Listing 1: 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 compilers may reorder read and write operations that have NO dependencies.

Algorithm 1 P_i of Peterson’s solution (two processes 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
```

2.1.2 Hardware support

Theorem (176) Test-and-Set：

- 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 2: Shared variables of Test-and-Set solution.

- 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 2 P_i (Test-and-Set).

```
 1: function $P_i$
 2:     repeat
 3:         $waiting[i] := True$
 4:         $key := True$                                  ▷ Local variable.
 5:         while $waiting[i] \wedge key$ do
 6:             $key :=$ TEST-AND-SET($\&lock$)
 7:         end while
 8:         $waiting[i] := False$
 9:         Critical section.
10:         $j := i + 1 \pmod n$
11:         while $j \neq i \wedge \neg waiting[j]$ do             ▷ 找下一個想進入的 $P_j$。
12:             $j := j + 1 \pmod n$
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
```

2.1.3 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 3: 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 3 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 4 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：

- 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 4: Shared variables of First Reader/Writer problem.

Algorithm 5 Writer (First Reader/Writer problem).

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

Algorithm 6 Reader (First Reader/Writer 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 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 5: Shared variables of Second Reader/Writer problem.

Algorithm 7 Writer (Second Reader/Writer 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 8 Reader (Second Reader/Writer 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 = 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 6: Shared variables of The sleeping barber problem.

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

Algorithm 9 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 10 Customer.

```
function CUSTOMER
    repeat
        WAIT($mutex$)
        if $waiting < n$ then             ▷ 入店。
            $waiting := waiting + 1$
            SIGNAL($customer$)            ▷ Wake up barber.
            SIGNAL($mutex$)
            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：
  - 根據公式 (2)，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 7: Shared variables of The dining-philosophers problem.

Algorithm 11 P_i of Algorithm 1 (The dining-philosophers problem).

```
function $P_i$
    repeat
        WAIT($no$)
        Hungry.
        WAIT($chopstick[i]$)
        WAIT($chopstick[(i+1 \pmod 5)]$)
        Eating.
        SIGNAL($chopstick[i]$)
        SIGNAL($chopstick[(i+1 \pmod 5)]$)
        Thinking.
        SIGNAL($no$)
    until $False$
end function
‘’
```

- Algorithm 2：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 3：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 8: Shared variables of The dining-philosophers problem.

Algorithm 12 *wait*(c) (counting semaphore).

```
function WAIT($c$)
    WAIT($s_1$)
    $c := c - 1$
    if $c < 0$ then
        SIGNAL($s_1$)
        WAIT($s_2$)                                   ▷ Process 卡住。
    else
        SIGNAL($s_1$)
    end if
end function
```

Algorithm 13 *signal*(c) (counting semaphore).

```
function SIGNAL($c$)
    WAIT($s_1$)
    $c := c + 1$
    if $c \leq 0$ then                          ▷ 先前有 process 卡住。
        SIGNAL($s_2$)
    end if
    SIGNAL($s_1$)
end function
```

Theorem () Non-busy waiting semaphore：

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

Listing 9: Non-busy waiting semaphore.

Algorithm 14 *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 15 *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 16 *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
        Enable interrupt.
    end if
end function
```

Algorithm 17 *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 (2.1.3) 中的 `Enable interrupt.` 和 `Disable interrupt.` 分別改為 `Entry section.` 和 `Exit section.` 並使用 Test-and-Set (2) 或 Compare-and-Swap 實現。

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

Algorithm 18 *wait*(S) of Algorithm 3 (disable interrupt design and busy waiting).

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

Algorithm 19 *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 (2.1.3)，將 Algorithm 3 (2.1.3) 中的 `Enable interrupt.` 和 `Disable interrupt.` 分別改為 `Entry section.` 和 `Exit section.` 。

2.1.4 Monitor

Theorem (189)

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 10: Data structure (The dining philosophers problem (Monitor)).

Algorithm 20 *pickup*(i).

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

Algorithm 21 *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 22 *putdown*(i).

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

Algorithm 23 *initialization_code*$()$.

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

Algorithm 24 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 {
    bool pr[3];
    Condition x;
}
```

Listing 11: Data structure of example of monitor

Algorithm 25 *Apply*(i).

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

Algorithm 26 *Release*(n).

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

Algorithm 27 *initialization_code*().

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

Algorithm 28 P_i of example of monitor.

```
function $P_i$
    PRINTERALLOCATION $pa$                         ▷ Shared variable.
    $n := pa$.APPLY($i$)
    Using printer $pr[n]$.
    $pa$.RELEASE($n$)
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 12: 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 29 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 30 $x.wait$.

```
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 31 $x.signal$.

```
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 (223)

- Dynamic binding 由 MMU 負責。
- Dynamic loading 由 programmer 負責，OS 無負擔。
- Dynamic linking 需要 OS 支持。
- 必須支援 dynamic binding 才可以在 execution time compaction。

Theorem (253) 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。

Theorem () Dirty bit：

- MMU：from 0 to 1.
- OS：from 1 to 0.

Theorem (263)

$$\text{TLB reach} = \text{TLB entries} \times \text{Frame size} \tag{7}$$

Theorem ((343)42, (344)44)

- Solaris ZFS uses **checksums** to provide fault-tolerance in case pointers are wrong.
- NFS:
  - Using RPC for remote file operations.
  - Writing to a file by a user are immediately visible to other users, since it does **NOT** support session semantics.
  - Does **NOT** support `open()` and `close()` operations.
  - Each request must provide a full set of arguments.
  - Supported file operations must be idempotent.
  - **NO** special measures are needed to recover a server from crash.

Theorem (309)

- Seek time：head 移到 **track** 的時間。

- Latency (Rotation) time：**sector** 移到 head 的時間。

Theorem () Storage：

- Smartphones normally do **NOT** have HDDs.
- Secondary storage is normally **non-volatile**.
- Wearable devices are normally equipped with **hard disks** to increase its storage space.

Theorem () Disk：

- **High-level** formatting creates a file system on a disk partition.
- A disk sector contains a header, a data area, and a trailer.
- In UNIX, disk scheduling algorithm is performed in the **disk driver**.
- A file system can be created across **multiple disk partitions**.
- **Disk device driver** can **NOT** be paged out, but page tables, memory-mapped files, shared memory can.
- Moving files between directories on the **same** disk partition and **deleting** files on a hard disk cause little overhead, but moving files between directories on **different** disk partitions cause much.
- The variation of disk I/O **latencies** under SSTF can be very high.

Theorem () Cybersecurity：

- Trojan Horse is a code segment that **misuses** its environment.
- Installing antivirus software is **NOT** an example of least privileges.
- Many routers are equipped with **firewall** and **VPN** functions.
- (109NYCU-10) Via HTTPS, ISPs can know the browsing website, but can **NOT** know the content by looking at the netwrok traffic.

Theorem () Cryptography：

- Public-key (asymmetric) cryptography 提供 digital signature 功能。

- AES：Symmetric, block cipher.
- DES：Symmetric, block cipher.
- RC4：Symmetric, stream cipher.
- RSA：Asymmetric，只要鑰匙夠長，沒有任何可靠的攻擊方法。
  - Authentication：將 message 與 hash 過再用 private key 加密的 message 串接。e.g. $M||\{h(M)\}_{K_{sa}}$.
  - Confidentiality：將用 one-time AES key 加密的 message 與用 public key 加密的 one-time AES key 串接。e.g. $\{M\}_{K_{da}}||\{K_{da}\}_{K_{pb}}$.
  - Confidentiality and authentication：將 authentication 的內容用 one-time AES key 加密，再與用 public key 加密的 one-time AES key 串接。e.g. $\{M||\{h(M)\}_{K_{sa}}\}_{K_{da}}||\{K_{da}\}_{K_{pb}}$.
- Digital certificate contains **private key** signed by the user.

Theorem () Kernel：

- Monolithic：UNIX, UNIX-like, Windows 9x, Android.
- Microkernel：Mach.
- Hybrid：Windows NT, Windows XP, macOS.
- Kernel processes are **NOT** allocated through paging and virtual memory interface.
- A **non-preemptive** kernel is free from race conditions on kernel data structures.
- **Preemptive** kernel design can **NOT** prevent the deadlock problem with kernel data structures from occurring in the kernel.
- Linux kernel is a **preemptive** kernel and a process running in a kernel mode could **NOT** be preempted.

Theorem () UID：

- Real UID: identify the real owner of the process and affect the permissions for sending signals.
- Effective UID: used for most access checks, including creating and accessing to a file.

- Saved UID: used when a program running with elevated privileges needs to do some unprivileged work temporarily.

Theorem () I/O：

- Buffered I/O: Read one block to cache when R/W, then copy from cache and return to reduce number of system call. Totally 2 copy operations.
- Unbuffered I/O: Directly transfer from disk without caching. Caching is conducted by the application. Number of copy operations is determined by the transfering method, and it's only 1 copy operation for block-transfering.
- A program using asynchronous I/O system calls in **NOT** simpler to write than using synchronous I/O system calls.

Theorem () File system：

- devfs：**Virtual** fs。一個 file 一個 device，但該 device 未必存在，**不確定 device mapping**。
- sysfs：**Virutal** fs。將 real connected devices 組織成**分階層**的 file directory，每個 device 有**唯一**對應的 directory。
- Device tree：每個 node 用 key 對應 value 方式紀錄 device properties，其中 value 可為空。

Theorem () GCD (Grand Central Dispatch)：

- 自動利用更多 CPU cores。
- 自動管理 thread life cycles。
- Move thread pool out of hand of developers and closer to OS.
- Dispatch tasks 時，可分在相同或不同 queues，分別稱作 serial 和 concurrent。Queues 間可分為 sync 和 async，前者同時間只允許一個 queue 執行，後者允許多個 queues 執行。

Theorem () Container：

- 所有 containers 共用 host OS。
- 相較 VM，不須打包 OS 就能執行，速度較快且空間小。

Theorem () MBR, BIOS, GPT, UEFI：

- BIOS 無法辨識 GPT（GUID Partition Table）。
- UEFI 用來定義 OS 和 firmware 間的 software interface。
- UEFI 是用模組化，動態連結的形式構建的系統，較 BIOS 而言更易於實現，容錯和糾錯特性更強，縮短了系統研發的時間。
- UEFI（Unified Extensible Firmware Interface）預啟動時就 load OS，且可以同時識別 MBR 和 GPT。
- GPT 使用 LBA（Logical Block Address）取代早期 CHS（Cylinder-head-sector）定址方式。
- GPT 的分割區表的位置資訊儲存在 GPT header 中，但第一個磁區仍然用作 MBR，之後才是 GPT header。

Theorem () Thread：

- **Native Windows threads** cause a user-mode to kernel-mode.
- Hyper-threading is **superscalar** and it can speedup **context switching**.
- Each thread of the program receives a **larger** CPU time with **many-to-one** thread model.
- Most operating systems **downgrade** the thread priority when it runs out of time quantum, but **boost** the priority when it returns from an I/O request.

Theorem () Cache：

- Physical caches do NOT flush at **context switching**.
- The TLB cache may require a flush after a page table update.
- Cache memories are usually hardware controlled, and OS may **NOT** even need to know their existence.

Theorem () Allocation：

- There is **NO** optimum solution to allocate contiguous memory from free holes.
- Extent allocation uses **contiguous physical** blocks, and it also needs defragmentation.

- Contiguous allocation offers the best R/W performance for **large** files.

Theorem () Page table：

- (**FALSE**) Use of shared memory can reduce the number of page table entries.
- (**FALSE**) The page table of Linux process is managed by the C runtime library (.so) in the process.
- (109NYCU-8d) For the **unused regions** in the virtual address space, the space overhead of the corresponding **page table entries** can be negligible.

Theorem () CPU scheduling：

- FIFO can outperform LRU.
- FIFO may have Convoy effect, which causes low **I/O** utilization.
- After making system calls, the process is still in running state.
- (**FALSE**) In a time-sharing system, a process does **NOT** leave running state unless it terminates or is preempted through a timer interrupt.

Theorem () Synchronization：

- Test-and-Set still wastes cycles when a process can **NOT** acquire a lock.
- To use shared memory, several system calls have to be invoked.
- Test-and-Set can be implementated in **user space**, provided that the lock variable is in a shared memory region.
- **Two-phase locking protocol (2PL)** ensures **conflict serializability**, but it may result in **deadlock**.
- (109NYCU-4c) OS needs to estimate MAX when a process enters **running** queue.

Theorem () Networks:

- Port：
  - 0: reserved.
  - $1 \sim 1023$: reserved by system, can be used by root.

  - 1024 ∼ 4999: used by client.
  - 5000 ∼ 65535: used by server.

- IP:
  - IPv6 routers can **NOT** perform a fragmentation.
  - Headers length: IPv4: 20B ∼ 60B; IPv6: fixed 40B.
  - IPv6 does **NOT** support broadcast, replaced by multicast.
  - ARP is used to discover a MAC address, associated with a given IP address.
  - IPSec can be used in VPNs.
  - ICMP messages can used to diagnose errors in IP operations.

Theorem ()

- Data fault: Access invalid data memory, which is signaled by **MMU**.
- NUMA is intrinsic in Von Neumann's computer model.
- `kmalloc`: physically contiguous; `vmalloc`: virtually contiguous; `malloc`: no constraints.
- `strncpy` 相較 `strcpy` 安全，且需要**預留一格**，可防止 buffer overflow。
- Java **interprets** Java bytecode operations **one at a time**.
- CLR, which is the implementation of .NET VM, **compiles** Microsoft intermediate language instructions **one at a time**.
- Normal instructions for the VM can execute **directly on the hardware** and **only the privileged instructions** must be simulated.
- Named pipes are referred to as **FIFOs** in UNIX systems. Once created, they appear as typical **files** in the file systems.
- Permission bits are stored at **inodes**.
- Five classic components: datapath, control unit, memory, input, and output.
- Data center cares more about **throughput** than response time.
- (109NYCU-6a.6b) Memory blocks on the **stacks** can **NOT** be freed at any time, but **heaps** can.

- **Stack** is good for locality.
- (**FALSE**) Programs written in different assmebly languages can ONLY be executed on specific hardware.
- Computer system can be divided into four components including hardware, OS, application programs, and users.
- Normal instructions for the virtual machines can execute directly on the hardware and ONLY the privileged instructions must be simulated.
- Bitmap is NOT a file.
- data section 存 global 和 static variables。
- When the block size is very large, the **spatial locality** within the block is lower.

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.