2025/04/16 4.8 4.14 4.20 6.7 6.18 111590004 資工三

5.12 5.17 5.22 5.25 6.15 張育昌

4.8 Provide two programming examples in which multithreading does *not* provide better performance than a single-threaded solution.

任何有關順序執行的程式，都不適合用多執行緒

① 個人報稅的程式

② Shell program

因為要監控自身工作環境、開啟的目錄、環境變數及工作目錄等。

4.14 Using Amdahl's Law, calculate the speedup gain for the following applications:

- 40 percent parallel with (a) eight processing cores and (b) sixteen processing cores
- 67 percent parallel with (a) two processing cores and (b) four processing cores
- 90 percent parallel with (a) four processing cores and (b) eight processing cores

Amdahl's law

$$\text{speedup} \le \frac{1}{S + \frac{(1-S)}{N}}$$

① 40% parallel, 60% serial

(a) 8 cores

$\frac{1}{0.6 + \frac{0.4}{8}} \approx 1.538$ #

gain (53.8%) #

(b) 16 cores

$\frac{1}{0.6 + \frac{0.4}{16}} = 1.6$ #

gain (60%) #

② 67% parallel, 33% serial

(a) 2 cores

$\frac{1}{0.33 + \frac{0.67}{2}} \approx 1.503$

gain (50.3%)

(b) 4 cores

$\frac{1}{0.33 + \frac{0.67}{4}} \approx 2.01$ #

gain (101%) #

③ 90% parallel, 10% serial

(a) 4 cores

$\frac{1}{0.1 + \frac{0.9}{4}} \approx 3.076$ #

(207%) gain

(b) 8 cores

$\frac{1}{0.1 + \frac{0.9}{8}} \approx 4.07$ #

(307%) gain #

4.20 Consider a multicore system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be greater than the number of processing cores in the system. Discuss the performance implications of the following scenarios.

a. The number of kernel threads allocated to the program is less than the number of processing cores.

b. The number of kernel threads allocated to the program is equal to the number of processing cores.

c. The number of kernel threads allocated to the program is greater than the number of processing cores but less than the number of user-level threads.

(a). kernel thread < CPU core

(b). kernel thread = CPU core

(c). User-level > kernel > CPU core

(a) 由於 CPU core > kernel threads，所以 kernel threads 的數量是 bottle neck 會導致 CPU core 無法完全發揮功能。

(Kernel threads 是滿的但 processing cores 閒置)

(b) 理想狀況，processing cores 可以被充分利用，(當 1 core 配 1 threads 時)。但發生阻塞後 cores 被閒置，造成浪費。

(c) 由於 kernel threads > processing cores，processing cores 發生阻塞時可以執行換出，讓 kernel threads 接手執行，提升效率、利用資源。

5.12 Discuss how the following pairs of scheduling criteria conflict in certain settings.

a. CPU utilization and response time

b. Average turnaround time and maximum waiting time

c. I/O device utilization and CPU utilization

(a) CPU utilization & Response Time

預期上，我們會希望 CPU利用率越大
Response Time 越小

如果我們要降低 Response time，那麼要優先處理互動型的工作（I/O型），會降低 CPU利用率

反之，提高CPU利用率，優先處理CPU密集型工作，則會提高 Response Time.

(b) Average turnaround & maximum waiting time

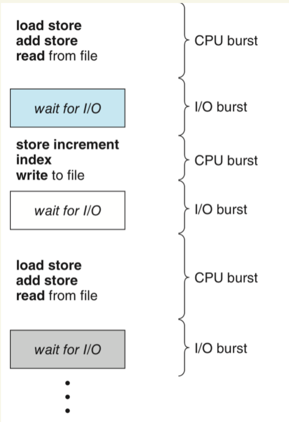
Average turnaround ⇓ 程式執行多久

maximum waiting time ⇓ 程式等多久

降低 average turnaround ⇒ 跑排程短的工作 ⇒ 導致排程長工作等很久

反之，若設置計時器讓排程長工作可以跑 會降低 maximum waiting time & 提高 average turnaround.

(c) I/O Device Utilization & CPU utilization



如左圖，要提 I/O利用率則 CPU的利用率會下降，反之若執行 CPU利用率高的程式則沒有行程處理 I/O，I/O利用率下降。

5.17 Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	5	4
P_2	3	1
P_3	1	2
P_4	7	2
P_5	4	3

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5, all at time 0.

a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

c. What is the waiting time of each process for each of these scheduling algorithms?

d. Which of the algorithms results in the minimum average waiting time (over all processes)?

(a) FCFS

P_1	P_2	P_3	P_4	P_5	
0	5	8	9	16	20

SJF

P_3	P_2	P_5	P_1	P_4	
0	1	4	8	13	20

Priority (數字大先，再來看順序)

P1	P5	P3	P4	P2

0 5 9 10 17 20

RR (Quantum = 2)

P1	P2	P3	P4	P5	P1	P2	P4	P5	P1	P4	P4

0 2 4 5 7 9 11 12 14 16 17 19 20

(b) Turnaround Time

	FCFS	SJF	Priority	RR
P1	5	13	5	17
P2	8	4	20	12
P3	9	1	10	5
P4	16	20	17	20
P5	20	8	9	16

(c) Waiting Time (燒一等)

	FCFS	SJF	Priority	RR
P_1	0	8	0	12
P_2	5	1	17	9
P_3	8	0	9	4
P_4	9	13	10	13
P_5	16	4	5	12

(D)

FCFS: $38/5 = 7.6$

SJF: $26/5 = 5.2$

Priority: $41/5 = 8.2$

RR: $50/5 = 10$

SJF have minimum avg waiting time

5.22 Consider a system running ten I/O-bound tasks and one CPU-bound task. Assume that the I/O-bound tasks issue an I/O operation once for every millisecond of CPU computing and that each I/O operation takes 10 milliseconds to complete. Also assume that the context-switching overhead is 0.1 millisecond and that all processes are long-running tasks. Describe the CPU utilization for a round-robin scheduler when:

a. The time quantum is 1 millisecond

b. The time quantum is 10 milliseconds

10 I/O-bound 1ms CPU 10ms I/O

1 CPU-bound

0.1ms Context switch

(a.) quantum 1ms

不管用哪個任務都有0.1ms的耗損

總共11個任務 11ms (10個I/O的CPU + 1個CPU)

11×0.1的context switching

$$\frac{11}{11+1.1} = \frac{11}{12.1} = \frac{110}{121} = \frac{10}{11} \times 100\% \approx 91\%$$

(b) 11個任務 20ms (10個I/O的CPU + CPU)

加上11×0.1的context switching

$$\frac{20}{20+1.1} = \frac{20}{21.1} \approx 94.8\%$$

5.25 Explain the how the following scheduling algorithms discriminate either in favor of or against short processes:

a. FCFS

b. RR

c. Multilevel feedback queues

a、先到先服務 (FCFS)

所以 long processes 先到就會把 short process 卡死。

b、RR

大家都有公平的執行時間，當然 short process 都會被先處理掉。較喜歡 short process

c、Multilevel Feedback Queues

以越短的 process 為主，時間越短的加入 priority 越前面的 Queue，會隨著等待時間越久將低 priority 的丟入高 priority.

偏好 short process。

6.7 The pseudocode of Figure 6.15 illustrates the basic push() and pop() operations of an array-based stack. Assuming that this algorithm could be used in a concurrent environment, answer the following questions:

a. What data have a race condition?

b. How could the race condition be fixed?

```
push(item) {
   if (top < SIZE) {
      stack[top] = item;
      top++;
   }
   else
      ERROR
}

pop() {
   if (!is_empty()) {
      top--;
      return stack[top];
   }
   else
      ERROR
}

is_empty() {
   if (top == 0)
      return true;
   else
      return false;
}
```

(a) top 及 stack[ ]
多個 process 會調用它們
導致資料讀取錯誤
或覆蓋

(b) 在 push() pop() 開始時加入互斥鎖 (mutex)
結束時解除互斥鎖，讓 stack
同時段只被一個 process 存取。

6.15 Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

使用者層級可以關閉 interrupt
可以關閉 timer interrupt
可以一直佔用 Processor，讓其它
程序無法執行。

6.18 The implementation of mutex locks provided in Section 6.5 suffers from busy waiting. Describe what changes would be necessary so that a process waiting to acquire a mutex lock would be blocked and placed into a waiting queue until the lock became available.

The main disadvantage of the implementation given here is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to `acquire()`. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU core is shared among many processes. Busy waiting also wastes CPU cycles that some other process might be able to use productively. (In Section 6.6, we examine a strategy that avoids busy waiting by temporarily putting the waiting process to sleep and then awakening it once the lock becomes available.)

The type of mutex lock we have been describing is also called a **spinlock** because the process "spins" while waiting for the lock to become available. (We see the same issue with the code examples illustrating the `compare_and_swap()` instruction.) Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. In certain circumstances on multicore systems, spinlocks are in fact the preferable choice for locking. If a lock is to be held for a short duration, one thread can "spin" on one processing core while another thread performs its critical section on another core. On modern multicore computing systems, spinlocks are widely used in many operating systems.

In Chapter 7 we examine how mutex locks can be used to solve classical synchronization problems. We also discuss how mutex locks and spinlocks are used in several operating systems, as well as in Pthreads.

為解決

busy waiting (忙等)

問題，我們可以

提供一個 queue

將 busy waiting 的 process 丟入 queue 中，重新被 CPU

喚醒時，將 process 狀態改成 就緒狀態。