4.8

1. 計算密集型任務：如果任務需要大量的計算而且你的系統只有一個核心，那麼多線程可能不會提高性能，因為單個核心一次只能執行一個任務。在這種情況下，單線程可能更有效，因為它避免了多線程之間的上下文切換開銷。

2. IO限制的任務：當任務主要受限於IO操作（例如磁盤讀寫、網絡通信）時，多線程可能並不總是提供更好的性能。這是因為IO操作通常會阻塞線程，使得多線程並行運行時，其中一些線程可能會因為等待IO而空閒。在這種情況下，更好的方法可能是使用非阻塞IO，或者使用非線程化的事件驅動模型。

4.10

(c) 全域變數 (Global variables) 是跨執行緒共享的程式狀態元件。

在多執行緒的處理中，每個執行緒都有自己的堆疊 (Stack memory) 和暫存器值 (Register values)。堆疊用於函式呼叫和本地變數，而暫存器存儲了處理器中的臨時數據。這兩者通常是執行緒私有的，每個執行緒都有自己的堆疊和暫存器值。

全域變數則存儲在程式的數據段中，這個段對於整個程序來說是共享的。因此，多個執行緒可以同時訪問和修改全域變數。這也是需要特別小心處理全域變數的並發編程中常見的問題之一，因為多個執行緒的同時訪問可能導致競爭條件 (Race condition) 和不一致的行為。

4.16

1. Input 和 Output 的執行緒數量：由於輸入和輸出都是在程序的起始和結束時進行的，並且只涉及單個文件的讀取和寫入，因此這兩個操作的性能不太可能受到多線程的影響。在這種情況下，一個執行緒來處理輸入和輸出應該足夠了，因為這些操作主要受限於磁盤IO速度，而不是CPU的計算能力。
2.
3. CPU-intensive 部分的執行緒數量：對於完全受CPU限制的部分，我們可以考慮創建多個執行緒來利用多核處理器的計算能力。但是，要注意在這個系統上，使用了一對一的執行緒模型，這意味著每個用戶執行緒都映射到一個內核執行緒。在這種情況下，如果創建過多的執行緒，可能會導致過多的內核執行緒競爭CPU資源，反而會增加上下文切換的開銷，從而影響性能。因此，建議將執行緒數量限制在與系統的CPU核心數相近的範圍內，這樣可以充分利用多核處理器的計算能力，同時避免過多的執行緒競爭CPU資源。

5.14

每個處理核心有自己的運行隊列：

優點：

- 簡單性： 每個核心維護自己的運行隊列，實現相對簡單，不需要考慮多個核心之間的同步和競爭問題。
- 效率： 沒有共享的運行隊列，不需要額外的同步機制和開銷，可以實現較高的效率。

缺點：

- 負載不均衡： 如果某個核心的運行隊列比其他核心更長，可能會導致負載不均衡。一些核心可能處於閒置狀態，而其他核心卻非常忙碌。
- 資源利用率不佳： 每個核心只能從自己的運行隊列中選擇進程運行，可能會導致系統中一些核心的資源被浪費。

所有處理核心共享單個運行隊列：

優點：

- 負載均衡： 所有核心共享一個運行隊列，進程可以在任何可用的核心上運行，從而實現較好的負載均衡。
- 資源利用率： 可以更好地利用系統中的資源，因為進程可以在任何可用的核心上運行。

缺點：

- 同步和競爭： 共享單個運行隊列需要額外的同步和競爭機制，可能會引入性能開銷，降低系統的效率。
- 複雜性： 管理和維護共享的運行隊列可能較為複雜，需要更多的設計和實現工作。

5.18

a.

```
Time: 0  10  20  25  35  45  55  65  65  70  75  90
       p2  p1  p2  p1  p3  p4  p3  p4  p5  p4  p6
```

完成順序 : p2 -> p1 -> p3 -> p4 -> p5 -> p6

b.

- p1: $35 - 0 = 35unit$
- p2: $25 - 0 = 25unit$
- p3: $65 - 20 = 45unit$
- p4: $75 - 25 = 50unit$
- p5: $70 - 45 = 25unit$
- p6: $90 - 55 = 35unit$

c.

- p1: $35 - (0 + 15) = 25unit$
- p2: $25 - (0 + 20) = 5unit$
- p3: $65 - (20 + 20) = 25unit$
- p4: $75 - (25 + 20) = 30unit$
- p5: $70 - (45 + 5) = 20unit$
- p6: $90 - (55 + 15) = 20unit$

5.22

在這個情況下，我們有十個I/O密集型任務和一個CPU密集型任務，我們使用循環調度器（Round-Robin Scheduler）。我們還假設I/O密集型任務每毫秒發出一次I/O操作，每個I/O操作需要10毫秒完成，上下文切換開銷為0.1毫秒，所有進程都是長時間運行的任務。

a.時間量子為1毫秒：

在這種情況下，每個I/O密集型任務在運行1毫秒後就會發出一次I/O操作。因此，每個I/O密集型任務需要11毫秒的時間才能完成一個循環（1毫秒的CPU運行 + 10毫秒的I/O等待）。在時間量子為1毫秒的情況下，一個時間量子的時間太短，無法完全執行一個I/O密集型任務的循環。當CPU切換到下一個任務時，這個I/O密集型任務可能仍然處於等待I/O操作完成的狀態。因此，CPU將花費大部分時間在上下文切換上，而不是在執行任務上，因此CPU利用率會很低。

b.時間量子為10毫秒：

在這種情況下，每個I/O密集型任務在運行10毫秒後才會發出一次I/O操作。因此，每個I/O密集型任務需要20毫秒的時間才能完成一個循環（10毫秒的CPU運行 + 10毫秒的I/O等待）。當時間量子增加到10毫秒時，CPU有足夠的時間來完整執行一個I/O密集型任務的循環，然後切換到下一個任務。因此，在這種情況下，CPU利用率將會提高，因為CPU主要用於執行任務，而不是在上下文切換上浪費時間。

5.25

先來先服務 (FCFS)

FCFS是一種非常簡單的排程算法，它將進程按照它們抵達系統的順序排隊，並且按照這個順序來執行。這意味著在FCFS中，短進程將被長進程阻塞，如果有一個長進程抵達系統並且在CPU上運行，則其他進程（包括短進程）必須等待它完成。因此，FCFS對於短進程沒有特別的偏好，並且可能導致長進程佔用CPU時間，而短進程則需要等待較長的時間。

循環調度 (RR)

循環調度（Round-Robin）是一種輪詢式排程算法，它將每個進程分配一個時間量子，進程在該時間量子內運行，如果這個時間量子用完了，則進程被放回到就緒隊列的末尾。因此，短進程在RR中通常會表現較好，因為它們在進程隊列中不需要等待太長的時間就能獲得CPU時間。但是，如果時間量子設置得太小，對於長進程來說，它們需要花費較多的時間在上下文切換上，這可能會影響長進程的性能。

多級反饋隊列 (Multilevel Feedback Queues)

多級反饋隊列是一種複雜的排程算法，它將進程分成不同的優先級別，並為每個級別分配不同的時間量子。通常情況下，短進程被分配到較高的優先級別，而長進程則被分配到較低的優先級別。這意味著短進程在MFQ中通常會被更快地執行，因為它們可以在較高的優先級別中獲得更多的CPU時間。如果一個進程在較低的優先級別上運行了一段時間，但沒有完成，它可以被提升到較高的優先級別，從而更快地完成。這使得MFQ對於短進程有較好的反應時間，並且能夠更好地處理長進程。

6.7

存在競爭條件的數據：

在並發環境中，push() 和pop() 操作涉及對堆棧中的共享數據進行讀取和寫入。如果多個線程同時執行push() 或pop() 操作，可能會導致競爭條件，從而導致數據不一致或其他錯誤。

如何解決競爭條件：

可以通過同步機制來解決競爭條件。例如，可以使用鎖（lock）或互斥鎖（mutex）來確保同一時間只有一個線程能夠訪問堆棧的共享數據。當一個線程執行push() 或pop() 操作時，它首先需要獲得鎖，然後執行操作完成後釋放鎖。這樣可以確保每個操作的執行是互斥的，從而避免了競爭條件。

6.15

在單處理器系統中，通過禁用中斷來實現同步原語在用戶級程序中使用是不適當的，原因如下：

系統可用性：

禁用中斷會影響整個系統的可用性。中斷在系統中具有重要的功能，例如用於處理硬件錯誤、處理IO操作等。如果禁用中斷，系統將無法正確地響應這些重要事件，可能導致系統出現嚴重的問題，甚至無法正常運行。

效能問題：

禁用中斷會導致系統無法及時地響應硬件事件，從而導致系統效能下降。特別是在單處理器系統中，中斷用於處理各種事件，包括IO操作、定時器事件等。禁用中斷會導致這些事件無法及時處理，影響系統的整體性能。

安全性問題：

禁用中斷可能導致系統存在安全漏洞。如果某個用戶級程序禁用了中斷並且出現了錯誤，可能會導致系統無法正常響應，從而增加了系統受攻擊的風險。

6.18

為了讓等待獲取互斥鎖的進程被阻塞並放入等待隊列，需要進行以下更改：

等待隊列：

需要維護一個等待隊列，用於存儲等待獲取互斥鎖的進程。這個等待隊列可以是一個先進先出（FIFO）的隊列或者其他適合的數據結構。

阻塞進程：

當一個進程試圖獲取一個被鎖定的互斥鎖時，如果該鎖已經被其他進程持有，則該進程應該被阻塞而不是進行忙等待。阻塞一個進程的方法取決於操作系統的具體實現，可能涉及將進程從運行狀態轉換為等待狀態。

鎖的獲取和釋放：

當持有互斥鎖的進程釋放鎖時，應該從等待隊列中選擇一個進程，將鎖分配給它。這樣，被阻塞的進程將有機會繼續執行，並獲得互斥鎖。