107 學年度 (2018 年秋季班) 作業系統 學期群組計畫需求說明書[*]

胡毓忠

國立政治大學資訊科學系

jong@cs.nccu.edu.tw

需求書提供日期：2018 年 9 月 18 日

1 注意事項

1. 各組請在 09/25 下午 4 點之前完成抽籤來選定學期群組計畫的實做題目。
2. 抽籤後所選定的實做題目不可改變與交換。
3. 請運用上課所學的 Multi-Thread Synchronization 概念與技術來完成群組計畫。雖然不限定所使用的程式語言如 Python, C, C++, Java, Scala, etc 但是必須要充分運用 Multi-Thread 的技術並且透過 Semaphore, Lock, or Monitor 等 Synchronization Primitives 來完成。
4. 所有應用問題中 Multi-Thread Synchronization 所需要同步的 entity (or object, thread) 的請用 Possion Process 來動態產生，也就是 entity 與 entity 之間進入系統的隨機時間差為指數分配函數 (Exponential Distribution)。如果可以請將主要參數以變數的方式來引入並且同步的 Multi-Thread 可以以 daemon 方式不中斷循環方式來執行。
5. 群組計畫分為兩部分實做部分佔 85%, 學期報告佔 15%, 各分組在 30 分鐘測試時（助教 20 分鐘老師 10 分鐘）請提供原始程式碼方便檢驗與提問。
6. 群組計畫測試時，請明確說明你需要多少 Threads（變動或固定）以及這些 Threads 所扮演的角色與功能為何？哪一些共有資源與共享變數 (Shared Variables) 是需要 Multi-Thread Synchronization 的 primitives 來完成同步？

[*]題組依據來源：Berztiss, Alfs T., Synchronization of processes, Department of Computer Science, University of Wollongong, Working Paper 82-11, 1982, 66p., http://ro.uow.edu.au/compsciwp/28

7. Multi-Thread Synchronization 過程至少各 Thread 以文字顯示同步運作狀態與結果，以確認 Multi-Thread Synchronization 的運作時過程正確無誤，如此才可以得到實做部分 80%，配上述文字的說明完成正確圖形使用者介面 (GUI) 顯示可以得到實做部分的 20%。另外提供額外實做與分析功能則可以再加分，例如進行 Multi-Thread Multi-Core 實做與觀察並提出相關心得與評論，使用 GitHub/Cloud Computing Platform 偕同計畫執行完整度也列入額外加分的考量。

8. 群組成績和個人成績的配分比是總成績的 20% 中群組成績佔 15%，個人成績佔 5%(註：個人成績評量請利用 Github 以及 Piazza 的討論紀錄來確認個人貢獻度)。

9. 請各組在 2018/12/01 之前完成 30 分鐘測試時段的安排，並在 2019/01/15（期末考日）最後期限前完成所有分組 30 分鐘測試與期末報告繳交。

10. 期末分組測試要項與報告格式與所需內容將在計畫測試前公佈。

2 Multi-Thread 同步的應用問題

1. **Tobacco Smokers (TS) Problem**

   Three smokers sit around a table. Each has a permanent supply of precisely one of three resources, namely tobacco, cigarette papers, and matches, but is not permitted to give any of this resource to a neighbor. An agent occasionally makes available a supply of two of the three resources. The smoker who has the permanent supply of the remaining resource is then in a position to make and smoke a cigarette. On finishing the cigarette this smoker signals the agent, and the agent may then make again available a supply of some two resources.

   The smokers are three threads, and the agent can be regarded as a set of three threads. As regards the latter, either none or exactly two of them run at anyone time. The problem is to have the six threads cooperate in such a way that deadlock is prevented, e.g., that when the agent supplies paper and matches, it is indeed the smoker with the supply of tobacco who gets both, instead of one or both of these resources being acquired by the other two smokers.

2. **The Sleeping Barber (SB) Problem**

   The barber shop has m barbers with m barber chairs, and n chairs ($m < n$) for waiting customers, if any, to sit in. If there are no customers present, a barber sits down in a barber chair and falls asleep. When a customer arrives, he has to wake up a sleeping barber. If additional customers arrive while all barbers are cutting customers' hair, they either sit down (if there are empty chairs) or leave the shop (if all chairs are full). The thread synchronization problem is to program the barbers and the customers without getting into race conditions.

3. **Readers and Writers (RW) Problem**

   Here one has a system of r readers and w writers that all access a common database (or some other resource). A reader may share the resource with an unlimited number of other readers, but a writer must be in exclusive control of the resource. We call this the RW problem. Two additional constraints characterize variants of the problem. Find a solution to each of the following RW1 and RW2 problem, which does not cause starvation of readers and writers.

   (a) Problem RW1. As soon as a writer is ready to write, no new reader should get permission to run. Starvation of readers is a possibility here.

   (b) Problem RW2. No writer is permitted to start running if there are any waiting readers. Here it is possible to starve the writers.

4. **Elevator Customer Scheduler (ECS) Problem**

   You've been hired by the University to build a controller for an elevator, using semaphores or condition variables. The elevator is represented as a thread; each student or faculty member is also represented by a thread. In addition to the elevator manager, you need to implement the routines called by the arriving student/faculty: $ArrivingGoingFromTo(intatFloor, inttoFloor)$. This should wake up the elevator, tell it the current floor a person is on, and wait until the elevator arrives before telling it which floor to go to. The elevator is amazingly fast, but it is not instantaneous it takes only 100 ticks to go from one floor to the next. Use $interrupt- > OneTick())$[1]

   You assume that there's only one elevator, and more than one person (there is no upper limit) can be in the elevator at a time. The trivial solution of serving one person at a time and putting others on hold, is not acceptable.

5. **Banker's Problem (BP)**

   A banker has a finite amount of capital, expressed in, say, kronor. The banker enters into agreements with customers to lend money. A borrowing customer is a thread. The following conditions apply:

   (a) The thread is created when the customer specifies a "need", i.e., a limit that his indebtedness will never be permitted to exceed.

   (b) The thread consists of transactions, where a transaction is either the advance of a krona by the banker to the customer, or the repayment of a krona by the customer to the banker.

   (c) The thread ends when the customer repays the last krona to the banker, and it is understood that this occurs within a finite time after the creation of the thread.

   (d) Requests for an increase in a loan are always granted as long as the current indebtedness is below the limit established at the creation of the thread, but the customer may experience a delay between the request and the transfer of the money.

   Here a means has to be found for the banker to determine whether the next payment of a krona to a customer creates the risk of deadlock.

[1]Implement an "alarm clock" class. Threads call "$Alarm : GoToSleepFor(inthowLong)$" to go to sleep for a period of time. The alarm clock can be implemented using the hardware Timer device (cf. timer.h). When the timer interrupt goes off, the Timer interrupt handler checks to see if any thread that had been asleep needs to wake up now. There is no requirement that threads start running immediately after waking up; just put them on the ready queue after they have waited for the approximately the right amount of time.

6. **Swimming Pool (SP) Problem**

The problem here is to synchronize the arrivals and departures at a swimming pool facility. There are two classes of resources, both in limited supply, n dressing rooms (or cubicles) and k baskets (where generally $n < k$). The thread that a bather goes through:

(a) Find available basket and cubicle.

(b) Change into swimwear and put one's street clothes in the basket.

(c) Leave cubicle and deposit the basket with the attendant.

(d) Swim (the pool is assumed to have unlimited capacity).

(e) Collect one's basket from the attendant.

(f) Find free cubicle and change back into street clothes.

To increase the degree of possible concurrency it helps to decompose these operations.
Thus (a) and (b) become:
a1. Find available cubicle
b1. Change into swimware
a2. Find available basket
b2. Put street clothes into basket

Similarly (f) becomes:
f1. Find free cubicle and empty the basket (thus making the basket available to someone else).
f2. Change into street clothes.

Now, however, it is possible to have deadlock: Arrivals occupy cubicles waiting for baskets to become available, but in so doing lock out prospective departures from the cubicles, thus preventing baskets from becoming available.

7. **Single Lane Bridge (SLB) Problem**

The problem is depicted in below figure. A bridge over a river is only wide enough to permit a single lane of traffic. Consequently, cars can only move concurrently if they are moving in the same direction. A safety violation occurs if two cars moving in different directions enter the bridge at the same time.

In our concurrent programming model, each car is a thread and the problem is to ensure that cars moving in different directions (eastbound and westbound) cannot concurrently access the shared resource, i.e., the bridge. The car is moving fast, but it is not instantaneous it might takes a random number of ticks to go from one side to the other. Again (Use *interrupt –> OneTick()*) to make the simulation more realistic, we must also ensure that cars moving in the same direction cannot pass each other. The bridge is also not strong enough to hold more than m cars at a time.

Find a solution to this problem which does not cause starvation. That is, cars that want to get across should eventually get across. However, we want to maximize use of the bridge. Cars should travel across to the maximum capacity of the bridge. If a car leaves the bridge going east and there are no westbound cars, then the next eastbound car should be allowed to cross. We don't want a solution which moves cars across the bridge m at a time, i.e., eastbound cars that are waiting should not wait until all m cars that are eastbound and crossing the bridge have crossed before being permitted to cross.