Operating System Semester Group Project Problem Set[1]

Fall Semester 2016, 2016/10/06

Due: 2016/01/17 (On-Site Demo and Report)

Department of Computer Science,

National Chengchi University

本群組計畫執行注意事項如下：

1. 各組請在助教所設定時間內完成抽籤來選定並確認學期計畫實做題目。
2. 抽籤後所選定與確認後的實做題目不可改變與交換。
3. 請運用上課所學的**Multi-Thread (or -Process) Synchronization**概念與技術來完成群組計畫，群組計畫實做不限定所使用程式語言,因此如**C, C++, Java, Python, Scala**等皆可，但是必須要明確的用**Multi-Thread (or -Process)**的技術並透過**Semaphore or Monitor**等**Synchronization Primitives**來完成**Threads (or Processes)**相互之間的同步與合作。
4. 群組計畫分為兩部分實做測試部分佔**80%**，學期書面報告佔**20%**，測試時請各組先提供簡式測試報告並在**2017/01/17**下午**05：00**之前完成最後完整電子檔繳交給課程助教。
5. 在群組計畫測試與報告中，請明確說明你需要多少類型的**Threads**（**or Processes**）以及這些不同類型的**Threads (or Processes)**其各自扮演的角色與功能為何？哪一些是共有資源與共享變數**(Shared Variables)**是需要進行**Multi-Thread (or -Process)**之間的同步與合作 。此外你必須要能明確標示這些**Threads (or Processes)**其各自的關鍵區域**(Critical Section)**的程式碼為何。
6. 所有單一類型**Thread (or Process)**所產生的個體數量可以彈性用參數的方式來動態設定，每一類型**Thread (or Process)**運作時其所產生動態個體**(entity)**數量的程序必須遵循**Poisson Process**,也就是個體與個體之間其產生的時間間隔分布 **(Inter-Arrival Time)**為指數分布函數**(Exponential Distribution)**。
7. **Threads (or Processes)**之間的同步運作可以用**Daemon**方式不間斷執行到某一個體達成某一數量或時間區間，除非使用者要求結束此應用系統操作並正常結束。
8. 請將**Multi-Thread (or –Process) Synchronization**的過程透過各**Thread (or Process)**以文字顯示運作狀態與結果，以確認**Multi-Thread (or Process) Synchronization**的運作過程正確無誤，如此則可以得到實做成績部分的**80%**，更進一步完成圖形使用者介面**(GUI)**的正確同步與合作則可以得到額外實做部分**20%**。
9. 我們提供額外實做與分析功能則可以再加分（**15%-25%**），例如進行**Multi-Thread (or –Process) Multi-Core**實做與觀察並提出具體心得與評論。
10. 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**14%**，個人成績佔**6%**，群組成績和個人成績的評定除了依據上述**3-9**的評分要項之外，我們將會參考各組使用**GitHub**分工合作的記錄檔來判定。
11. 請在本課程期末考（**2016/01/10**）完畢後的一星期之內（**2016/01/17**）完成各組的分組測試，各組測試流程是先經過助教**20**分鐘的初步測試再由老師**10**分鐘確認最後評量結果。各組測試時必須要全員到齊，缺席者視同沒有應考，其學期群組計畫將以**0**分計算。

---

[1] 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

1. 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 the problem, e.g., (1) as soon as a writer is ready to write, no new reader should get permission to run. Starvation of readers is a possibility here. (2) No writer is permitted to start running if there are any waiting readers. Here it is possible to starve the writers. You must consider how to solve the readers or writers starvation problems.

2. 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.

3. Banker's Problem (BP)

A banker has a finite amount of capital, expressed in, say, dollar. 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 dollar by the banker to the customer, or the repayment of a dollar by the customer to the banker.
c. The thread ends when the customer repays the last dollar 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 dollar to a customer creates the risk of deadlock.

4. 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 swimwear

a2. Find available basket

b2. Put street clothes into basket

Similarily (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.

5. 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 (int atFloor, int toFloor)*. 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()* [2])

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.

[2] Implement an "alarm clock" class. Threads call "Alarm:GoToSleepFor(int howLong)" 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. 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.