Sample Synchronization Problems
CS 4410/4411, Spring 2010
Emin Gün Sirer

1. You have been hired to coordinate people trying to cross a river. There is only a single boat, capable of holding at most three people. The boat will sink if more than three people board it at a time. Each person is modeled as a separate thread, executing the function below:

```
Person(int location)
// location is either 0 or 1;
// 0 = left bank, 1 = right bank of the river
{
     ArriveAtBoat(location);
     BoardBoatAndCrossRiver(location);
     GetOffOfBoat(location);
}
```

Synchronization is to be done using monitors and condition variables in the two procedures `ArriveAtBoat` and `GetOffOfBoat`. Provide the code for `ArriveAtBoat` and `GetOffOfBoat`. The `BoardBoatAndCrossRiver()` procedure is not of interest in this problem since it has no role in synchronization. `ArriveAtBoat` must not return until it safe for the person to cross the river in the given direction (it must guarantee that the boat will not sink, and that no one will step off the pier into the river when the boat is on the opposite bank). `GetOffOfBoat` is called to indicate that the caller has finished crossing the river; it can take steps to let other people cross the river.

Answer

```
Class Boat:
     def __init__(self):
          self.boatlocation = True; // T for left bank, F for right
          self.boatcount = 0;
          self.mutex = Lock()
          self.leftbank =Condition(self.mutex)
          self.rightbank=Condition(self.mutex)

def ArriveAtBoat(self, location):
  with self.mutex:
    mybank = self.leftbank if location else self.rightbank
    while self.boatlocation != location or self.boatcount == 3:
     mybank.wait()

def GetOffOfBoat(self, location):
  with self.mutex:
     self.boatcount -= 1
     if self.boatcount == 0:
          // we arrived on the other side, update boat direction
          self.boatlocation = not self.boatlocation;
          newbank = self.leftbank if location else self.rightbank
          newbank.notifyAll()
```

Note that this solution may lead to starvation. The readers is encouraged to develop a starvation-free solution.

2. Ping and pong are two separate threads executing their respective procedures. The code below is intended to cause them to forever take turns, alternately printing “ping” and “pong” to the screen. The `minithread_stop`() and `minithread_start` () routines operate as they do in your project implementations: `minithread_stop`() blocks the calling thread, and `minithread_start` () makes a specific thread runnable if that thread has previously been stopped, otherwise its behavior is unpredictable.

```
void ping()
{
     while(true) {
          minithread_stop();
          Printf(“ping is here\n”);
          minithread_start(pongthread);
     }
}
void pong()
{
     while(true) {
          Printf(“pong is here\n”);
          minithread_start(pingthread);
          minithread_stop();
     }
}
```

a. The code shown above exhibits a wellknown synchronization flaw. Briefly outline a scenario in which this code would fail, and the outcome of that scenario.
b. Show how to fix the problem by replacing the `minithread_stop and start` calls with semaphore P and V operations.
c. Implement ping and pong correctly using a mutex and condition variables.

```
Sema sping, spong = sema_init(0);
void ping()
{
     while(true) {
          P(sping);
          Printf(“ping is here\n”);
          V(spong);
     }
}
void pong()
{
     while(true) {
          Printf(“pong is here\n”);
          V(sping);
          P(spong);
     }
}
```

```
classPingPong:
def __init__(self):
     self.m = Lock()
     self.turn = True
     self.cond = Condition(self.m)

def ping(self):
  with self.m:
     while not self.turn:
          self.cond.wait()
     print “ping is here”
     self.turn = False
     self.cond.notify()


def pong(self):
  with self.m:
     while self.turn:
          self.cond.wait()
     print “pong is here”
     self.turn = False
     self.cond.notify()
```

3. You are designing a data structure for efficient dictionary lookup in a multithreaded application. The design uses a hash table that consists of an array of pointers each corresponding to a hash bin. The array has 1001 elements, and a hash function takes an item to be searched and computes an entry between 0 and 1000. The pointer at the computed entry is either null, in which case the item is not found, or it points to a doubly linked list of items that you would search sequentially to see if any of them matches the item you are searching for. There are three functions defined on the hash table: Insertion (if an item is not there already), Lookup (to see if an item is there), and deletion (to remove an item from the table). Considering the need for synchronization, would you:

1. Use a mutex over the entire table?
2. Use a mutex over each hash bin?
3. Use a mutex over each hash bin and a mutex over each element in the doubly linked list?

Justify your answer.

Answer: A mutex over the entire table is undesirable since it would unnecessarily restrict concurrency. Such a design would only permit a single insert, lookup or delete operation to be outstanding at any given time, even if they are to different hash bins. A mutex over each element in the doubly linked list would permit the greatest concurrency, but a correct, deadlock-free implementation has to ensure that all elements involved in a delete or insert operation, namely, up to three elements for an delete, or two elements and the hash bin for inserts/some deletes, are acquired in a well-defined order. A mutex over each hash bin is a compromise between these two solutions – it permits more concurrency than solution 1, and is easier to implement correctly than solution 2.

4. A car is manufactured at each stop on a conveyor belt in a car factory. A car is constructed from the following parts - chassis, tires, seats, engine (assume this includes everything under the hood and the steering wheel), the top cover, and painting. Thus there are 6 tasks in manufacturing a car. However, tires,

seats or the engine cannot be added until the chassis is placed on the belt. The car top cannot be added until tires, seats and the engine are put in. Finally, the car cannot be painted until the top is put on.

A stop on the conveyor belt in your car company has four technicians assigned to it - Abe, Bob, Charlie, and Dave. Abe is skilled at adding tires and painting, Bob can only put the chassis on the belt, Charlie only knows how to attach the seats, and Dave knows how to add the engine as well as how to add the top.

Write pseudocode for Abe, Bob, Charlie and Dave to be able to work on the car, without violating the task order outlined above. You can use semaphores, mutexes or monitors in your solution.

5. Write a simulator for Dragon Day, where each architect and engineer is modeled as a separate thread executing their respective functions.

```
void initsimulator() {
      // TODO: initialize any synch variables
}
void architecture_student() {

      // TODO: Architect enters equad
      EnterEngineeringQuad();

      // TODO: an architect should do nothing until s/he
     // observes a snowball thrown by an engineer
     WaitUntilSnowballIsThrown()

      // run out of the Equad, but only after seeing a snowball
     // thrown by an engineer
      ExitEngineeringQuad();
}
void engineer(){
      // TODO: an engineer should wait until there
      // are at least two architects who are not
      // running around.
      WaitUntilThereAreTwoArchitects();

      // throw two snowballs
      ThrowSnowball(); // first one
      ThrowSnowball(); // second one
}
```

Your task is to write `EnterEngineeringQuad, WaitUntilSnowballIsThrown, WaitUntilThereAreTwoArchitects, and ThrowSnowball().`

```
Answer:
EnterEngineeringQuad() { V(archies); }
WaitUntilSnowballIsThrown { P(snowballs); }
WaitUntilThereAreTwoArchitects() { P(archies); P(archies); }
ThrowSnowball() { V(snowballs); }
```

6. You have been hired by Large-Concurrent-Systems-R-Us, Inc. to review their code. Below is their **atomic_swap** procedure. It is intended to work as follows:

Atomic_swap should take two queues as arguments, dequeue an item from each, and enqueue each item onto the opposite queue. If either queue is empty, the swap should fail and the queues should be left as they were before the swap was attempted. The swap must appear to occur atomically – an external thread should not be able to observe that an item has been removed from one queue but not pushed onto the other one. In addition, the implementation must be concurrent – it must allow multiple swaps between unrelated queues to happen in parallel. Finally, the system should never deadlock.

Note if the implementation below is correct. If not, explain why (there may be more than one reason) and rewrite the code so it works correctly. Assume that you have access to enqueue and dequeue operations on queues with the signatures given in the code. You may assume that q1 and q2 never refer to the same queue. You may add additional fields to stack if you document what they are.

```
extern Item *dequeue(Queue *);      // pops an item from a stack
extern void enqueue(Queue *, Item *); // pushes an item onto a stack

void atomic_swap(Queue *q1, Queue *q2) {
      Item *item1;
      Item *item2; // items being transferred

      P(q1->lock);
      item1 = pop(q1);
      if(item1 !=  NULL) {
            P(q2->lock);
            item2 = pop(q2);
            if(item2 != NULL) {
                  push(q2, item1);
                  push(q1, item2);
                  V(q2->lock);
                  V(q1->lock);
            }
      }
}
```

Answer:
The code has three problems: it can deadlock, it fails to restore q1, and it has unmatched P’s and V’s.

```
void atomic_swap(Queue *q1, Queue *q2) {
      Item *item1;
      Item *item2; // items being transferred

      if(q1->id > q2->id) {
            // impose ordering on P operations
            Tmp = q1;
            q1 = q2;
            q2 = tmp;
      }
      P(q1->lock);
      P(q2->lock);
      item1 = dequeue(q1);
      if(item1 !=  NULL) {
            item2 = dequeue(q2);
            if(item2 != NULL) {
                enqueue(q2, item1);
                enqueue (q1, item2);
           } else {
                enqueue (q1, item1);
      }
      V(q2->lock);
      V(q1->lock);
}
```

7. Implement a synchronization barrier. Synchronization barriers are a common paradigm in many parallel applications. A barrier is supposed to block the calling thread until all N threads have reached the barrier. (Parallel apps often divide up the work, e.g. matrix operations or graphical rendering, among N processes, each of which compute independently, reach the barrier, exchange results, and then work on the next phase of computation).

Answer:
One use barriers are simple to implement with monitors and condition variables:

```
class Barrier:
def __init__(self, N):
      self.count = 0
      self.N = N
      self.lock = Lock()
      self.everyoneatbarrier = Condition(self.lock)

def barrier(self, processid):
  with self.lock:
      self.count += 1
      if self.count == self.N:
           self.everoneatbarrier.notifyAll()
      while self.count != self.N:
           self.everyoneatbarrier.wait()
```

Note that the code above is good for one use. If these threads are to reuse this barrier, we need to reset it to 0 when the last thread is out. Naïve attempts to accomplish this lead to errors, for instance, the following is a bad barrier implementation:

```
class Barrier:
def __init__(self, N):
      self.count = 0
      self.N = N
      self.lock = Lock()
      self.everyoneatbarrier = Condition(self.lock)

def barrier(self, processid):
  with self.lock:
      self.count += 1
      if self.count == self.N:
           self.everoneatbarrier.notifyAll()
      while self.count != self.N:
           self.everyoneatbarrier.wait()
      self.count = 0
```

If the last line is replaced with "`self.count -= 1`", the program is still buggy (you should be able to discern why). Here's a non-buggy implementation:

```
class Barrier:
def __init__(self, N):
     self.incount = 0
     self.outcount = 0
     self.N = N
     self.lock = Lock()
     self.everyoneatbarrier = Condition(self.lock)

def barrier(self, processid):
  with self.lock:
     self.incount += 1
     if self.incount == self.N
          self.everoneatbarrier.notifyAll()
     while self.incount < self.N or
          (self.incount >= self.N and self.outcount < self.N):
          self.everyoneatbarrier.wait()
     self.outcount += 1
     if self.outcount == self.N:
          self.outcount = 0
          self.incount -= self.N
```

Note that if we replace the last line with "`self.incount = 0`", we get a buggy implementation once again. You should be able to describe when and how the bug would be triggered.

Naïve attempts with semaphores are likely to contain errors, e.g.:

```
void barrier(int processid) {
     P(mutex);
     If(++count != N) {
          V(mutex);
          P(EveryoneHasReachedBarrier);
     } else {
          V(mutex);
          For(I = 0; I < N; ++I)
               V(EveryoneHasReachedBarrier);
     }
}
```

8. Implement a deadlock-free solution to the dining philosophers problem.

9. A system has four processes and five allocatable resources. The current allocation and maximum needs are as follows:

Process	Allocated	Maximum	Available
A	1 0 2 1 1	1 1 2 1 3	0 0 x 1 1
B	2 0 1 1 0	2 2 2 1 0	
C	1 1 0 1 0	2 1 3 1 0	
D	1 1 1 1 0	1 1 2 2 1	

What is the minimum value of x for which this is a safe state ?

10. Your program has the following structure: one thread sits in an event-processing loop, waiting for events to be added to a queue. If the queue is empty, it should block until it is non-empty; if it is non-empty, it should dequeue all the events on it and process them, then continue. Other threads in the program add events to the queue at arbitrary times. What synchronization is necessary to ensure a correct system?

```
Queue eventq;
void eventloop(void) {
      //TODO: synchronize
}
void addevent(Event *e) {
      //TODO: synchronize
      enqueue(eventq, e);
}
```

11. A certain bar is a well-known hangout for detectives. If a detective comes to the bar and there are no clients at the bar, the detective talks to the bartender. If one or more clients are present, the detective approaches the client who arrived earliest, and they leave the bar. If a client arrives and there are no detectives at the bar, the client orders a drink and waits. If there are one or more detectives, the client and the detective who arrived earliest leave the bar. What synchronization is necessary to ensure a correct system?

```
void detective(void) {
      //TODO: synchronize
}
void client(void) {
      //TODO: synchronize
}
```

12. The same setup, except that now when a detective arrives, if there are clients in the bar, the detective leaves with all of them. A client arriving when there are multiple detectives still leaves with only one.

13. A collection of threads are sending data over a network. A send operation uses a varying number of packet buffers, with no limit to the maximum number. There are N buffers in the system, each of which is either free or in use. If a thread executes a send that requires more buffers than there are free, then it blocks, otherwise it marks them as used and returns. A network thread is responsible for sending buffers and freeing them, which is not a very fast operation (it's a slow network). As buffers become free, they will be allocated to blocked threads first, so that eventually they will meet their requirement and become unblocked. What synchronization primitives are necessary to ensure correctness, and how should they be used?

```
void send(int numBlocksNeeded) {
      //TODO: synchronize
}
void packetsent(int numBlocksPacketFreed) {
      //TODO: synchronize
}
```

14. Day Hall is considering a new policy that will put an end to its long-standing practice of gender-based segregation in bathrooms, while upholding the age-old tradition that only people of the same gender can use a bathroom at the same time. The new rules postulate that a bathroom can be in three states: empty, women present, men present. When women are present, other women may enter but no men, and vice

versa. An empty bathroom may be used by a person of either gender, but becomes dedicated to that gender until the last person of that gender leaves. You can assume that bathrooms have infinite capacity, i.e. everyone who wants to enter can do so as long as the bathroom is empty or occupied by people of their own gender.

Write the following procedures: `woman_wants_to_enter`, `man_wants_to_enter`, `woman_leaves` and void `man_leaves`. You can call on the helper function “`int get_my_id()`” to get a unique identifier for that person, if necessary.