Homework Set 2 - Practical Skills and Concurrency

Yiming Liu 2023010747

Problem 1

Answer:

(1) `/^[\w.-]+@(mails?\.)?tsinghua\.edu\.cn$/`

(2) `/^[\w.-]+@(\w+\.)?tsinghua\.(edu|org)\.cn$/`

(3)`/^[\w.-]+@(?!mails?)(\w+)\.tsinghua\.edu\.cn$/`

(4)`sort departments.txt | uniq | wc -l`

Problem 2

Answer:

Scenario 1: After A creates a `newNode(A)` it switch to B, then B creates its own `newNode(B)`. B sets `tail` to `newNode(B)` and links `oldTail.next` to its node.

After B finishes, A sets `oldTail = tail`, which means that $A's$ `oldTail` is `newNode(B)`. Then A links `oldTail.next` to its `newNode(A)`. In this way, `head` points to `newNode(B)` and `newNode(B)` points to `newNode(A)`. So there's no problem.

Scenario 2: After A sets `oldTail = tail`, it switch to B. Then B creates its own `newNode(B)`. B sets `tail` to `newNode(B)` and links `oldTail.next` to its node.

Back to A, `oldTail` now is $B's$ `oldTail`. A links `oldTail.next` to its `newNode(A)`. In this way, `newNode(A)` replace `newNode(B)` and `newNode(B)` is lost.

Scenario 3: After A sets `tail = newNode(A)` it switch to B, then B creates its own `newNode(B)`. B sets `oldTail` to be `tail` which is `newNode(A)`. Then B sets `tail` to `newNode(B)` and links `oldTail.next` to its node. So `newNode(A)` points to `newNode(B)`.

After B finishes, A sets `oldTail.next = newNode(A)`. This `oldTail` is $A's$ `oldTail` so it's `head`. So `head` points to `newNode(A)` and `newNode(A)` points to `newNode(B)`. Hence there's no problem.

Problem 3

(1) No need to change the code. Because `lock.Acquire()` ensures that only one thread can modify the queue at a time (mutual exclusion).

(2)

```C
Lock lock;
Condition dataready;
Condition notFull;
int max_size;
Queue queue;
AddToQueue(item) {
  lock.Acquire(); //  Get Lock
  while (queue.size() == max_size) {
    notFull.wait(&lock);
```

```
  }
  queue.enqueue(item); // Add item
  dataready.signal() ; // Signal any waiters
  lock.Release(); //  Release Lock
}
RemoveFromQueue() {
  lock.Acquire(); /./ Get Lock
  while (queue.isEmpty()) {
    dataready.wait(&lock); // If nothing, sleep
  }
  item = queue.dequeue(); // Get next item
  notFull.signal();
  lock.Release(); // Release Lock
  return(item);
}
```

(3)

```c
ReadFromQueue() {
  lock.Acquire(); // Get Lock
  while (queue.isEmpty()) {
    dataready.wait(&lock); // If nothing, sleep
  }
  item = queue.read();
  lock.Release(); // Release Lock
  return(item);
}
```

Problem 4

```c
Semaphore lock = 1;
int owner = -1;
void RLock() {
  int cur_id = getMyTID();
  if (owner != cur_id) {
    lock.P();
    owner = cur_id;
  }
}
void RUnLock() {
  int cur_id = getMyTID();
  if (owner == cur_id) {
    owner = -1;
    lock.V();
  }
}
```

Problem 5

(1)

```c
Semaphore barberReady = 0;
Semaphore accessWaitRoomSeats = 1;
Semaphore customerReady = 0;
int numberOfFreeWaitRoomSeats = N;
void Barber () {
  while (true) {
    customerReady.P();
    accessWaitRoomSeats.P();
    numberOfFreeWaitRoomSeats += 1;
    accessWaitRoomSeats.V();
    cutHair(); // Cut customer's hair
    barberReady.V();
  }
}
void Customer () {
  accessWaitRoomSeats.P();
  if (numberOfFreeWaitRoomSeats > 0) {
    numberOfFreeWRSeats -= 1;
    customerReady.V();
    accessWaitRoomSeats.V();
    barberReady.P();
    getHairCut(); // Customer gets haircut
  } else {
    accessWaitRoomSeats.V();
    leaveWithoutHaircut(); // No haircut
  }
}
```

(2) `accessWaitRoomSeats` is used for mutex, and `customerReady` and `barberReady` are used for scheduling constraints.

(3) Customer A_0 is having haircut when customer B arrives and has many hair. B starts waiting, but every time A_i nearly finishes, A_{i+1} arrives and A_{i+1} has no hair at all. Thus, B will wait forever.

Deadlock will not happen because the barber will always be able to cut hair for the next customer.

The starvation will happen with less probability if the barber randomly selects a customer to cut hair.

We can arrange the order in the waiting room to avoid starvation. In other words, FIFO (First In, First Out) queueing discipline would be effective. Which means barber always pick the one who come earliest.