CS 431: Programming Lab

Java Assignment

Akul Agrawal
160101085

Problem 1

Synchronization for accessing the clothes by the multiple threads is done by using semaphores. A semaphore controls access to a shared resource through the use of a counter. Here, clothes are the shared resources and we have used individual semaphores to lock each of them and set the semaphore counter value equal to 1 i.e. each cloth can be picked up by only one robot thread.

Code Snippets:

ClothOrdering.java

```
int[] PickCloth() {
    int[] cloth = new int[3];
    int n = -1;
    try {
        semLock();
        if (Clothes.size() > 0) {
            n = rand.nextInt(Clothes.size());
        } else {
            cloth[0] = Constants.NULL_SOCK;
            cloth[1] = Constants.NULL_SOCK;
            cloth[2] = Constants.NULL_SOCK;
            semRelease();
            return cloth;
        }
        if (n < Clothes.size()) {
            cloth[0] = Clothes.get(n);
            cloth[1] = Num.get(n);
            cloth[2] = OrderID.get(n);
            Clothes.remove(n);
            Num.remove(n);
            OrderID.remove(n);
            semRelease();
            return cloth;
        }
    }
    catch (Exception e) {
        System.out.println(e);
    }
    semRelease();
    return PickCloth();
}
```

semLock() acquires semaphore lock on the shared data i.e. data structures Clothes, Num and OrderID so that no two threads/robots can pick the same clothes.

Clothes[i]: Stores type of cloth of order i

Num[i]: Stores quantity of cloth of order i

OrderID[i]: Stores order id of order i

OrderManager.java

```
void ManageOrder(int cloth, int num, int orderID) {
    try {
      if (cloth == Constants.SMALL) {
        semS.acquire();
        if (SmallTshirts >= num) {  // have sufficient clothes
          SmallTshirts -= num;    // num clothes
          System.out.println(String.format("Order %d successful", orderID));
          PrintInventory();
        } else { // don't have sufficient clothes of that color
          System.out.println(String.format("Order %d failed", orderID));
        }
        semS.release();
      } else if (cloth == Constants.MEDIUM) {
        semM.acquire();
        if (MediumTshirts >= num) {  // have sufficient clothes
          MediumTshirts -= num;    // num clothes
          System.out.println(String.format("Order %d successful", orderID));
          PrintInventory();
        } else { // don't have sufficient clothes of that color
          System.out.println(String.format("Order %d failed", orderID));
        }
      } else if (cloth == Constants.LARGE) {
        semL.acquire();
        if (LargeTshirts >= num) {  // have sufficient clothes
          LargeTshirts -= num;    // num clothes
          System.out.println(String.format("Order %d successful", orderID));
          PrintInventory();
        } else { // don't have sufficient clothes of that color
          System.out.println(String.format("Order %d failed", orderID));
        }
      } else if (cloth == Constants.CAP) {
        semC.acquire();
        if (Caps >= num) {  // have sufficient clothes
          Caps -= num;    // num clothes
          System.out.println(String.format("Order %d successful", orderID));
          PrintInventory();
```

```
            } else { // don't have sufficient clothes of that color
                System.out.println(String.format("Order %d failed", orderID));
            }
        }
    }
    catch (Exception e) {
        System.out.println(e);
    }
}
```

For each type of cloth, a semaphore lock is acquired to synchronize the thread for update on shared data i.e. number of clothes left of that type in the inventory. If synchronization is not done, the update may result in incorrect results.

For example, if two threads update the value of , say, caps, without synchronization, the two updates, i.e., Caps = Caps (initial) - num (thread1) and Caps = Caps (initial) - num (thread2) might take place at the same time. Thus, finally, either Caps = Caps (initial) - num (thread1) or Caps = Caps (initial) - num (thread2). While the actual value of Caps should be Caps = Caps (initial) - num (thread1) - num (thread2).

Problem 2

Packaging Unit and Sealing Unit are independent of each other in case no buffer is full, and both can work simultaneously. In absence of concurrency, while one of them is working, the other one has to wait for it to complete, even if it is free to do its job. So concurrency is required to allow packaging and sealing to take place efficiently.

Code Snippets:

ManufacturingUnit.java

```
void addToTray(int state, int n) {
    if (state == Constants.STATE_PACK) {
        if (n == Constants.B1) {
            try {
                synchronized (B1PackBuf) {
                    if (B1PackBuf == Constants.MAX_B1_PACK) {
                        B1PackBuf.wait();
                    }
                    B1PackBuf++;
                }
            }
            catch (Exception e) {
                System.out.println(e);
```

```
            }
        }
        else {
            try {
                synchronized (B2PackBuf) {
                    if (B2PackBuf == Constants.MAX_B2_PACK) {
                        B2PackBuf.wait();
                    }
                    B2PackBuf++;
                }
            }
            catch (Exception e) {
                System.out.println(e);
            }
        }

    }
    else {
        try {
            synchronized(SealBuf) {
                if (SealBuf.size() >= Constants.MAX_SEAL_BUF) {
                    SealBuf.wait(); //wait for the queue to become empty
                }
                SealBuf.put(n);
            }
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
  }

int PickBottle(int state) {
    int n = -1;

    if (state == Constants.STATE_PACK) {
        try {
            synchronized (B1PackBuf) {
                synchronized (B2PackBuf) {
                    synchronized (B1Unfin) {
                        synchronized (B2Unfin) {

                            if(Priority == 0) {
```

```
        Priority = 1;
        if(B1PackBuf > 0) {
            n = 2;
            B1PackBuf--;
            if(B1PackBuf == Constants.MAX_B1_PACK - 1)
                B1PackBuf.notify();
        }
        else if(B2PackBuf > 0) {
            n = 3;
            B2PackBuf--;
            if(B2PackBuf == Constants.MAX_B2_PACK - 1)
                B2PackBuf.notify();
        }
        else {
            n = choose(state, B1Unfin, B2Unfin);
            if(n == 0) {
                B1Unfin--;
            }
            else if(n == 1) {
                B2Unfin--;
            }
        }
    }
    else {
        Priority = 0;
        if(B2PackBuf > 0) {
            n = 3;
            B2PackBuf--;
            if(B2PackBuf == Constants.MAX_B2_PACK - 1)
                B2PackBuf.notify();
        }
        else if(B1PackBuf > 0) {
            n = 2;
            B1PackBuf--;
            if(B1PackBuf == Constants.MAX_B1_PACK - 1)
                B1PackBuf.notify();
        }
        else {
            n = choose(state, B1Unfin, B2Unfin);
            if(n == 0) {
                B1Unfin--;
            }
            else if(n == 1) {
```

```
                        B2Unfin--;
                    }
                }
            }

        }
    }
  }
  catch (Exception e) {
    System.out.println(e);
  }
}
else {
    synchronized (B1Unfin) {
    synchronized (B2Unfin) {

      synchronized(SealBuf) {
        if(SealBuf.size() > 0) {
          try {
            int m = SealBuf.take();
            if (SealBuf.size() == Constants.MAX_SEAL_BUF - 1) {
              SealBuf.notify();
            }
            n = 4 + m;
          }
          catch (Exception e) {
            System.out.println(e);
          }
        }
        else {
          n = choose(state, B1Unfin, B2Unfin);
          if(n == 0) {
            B1Unfin--;
          }
          else if(n == 1) {
            B2Unfin--;
          }
        }
      }

    }
```

```
        }
      }
      long endTime   = System.currentTimeMillis();
      long totalTime = endTime - startTime;
      if(((state == 0) && (totalTime > Time - 150)) || ((state == 1) && (totalTime > Time - 250)))
{
        n = -1;
      }
      return n;
   }
```

OrderManager.java

```
void Package(int n, boolean sealed) {
   synchronized (Packaged[n]) {
      synchronized (Godown[n]) {
         Packaged[n] += 1;
         if (sealed) {
            Godown[n] += 1;
         }
         // Packaging complete
      }
   }
}

void Seal(int n, boolean packaged) {
   synchronized (Sealed[n]) {
      synchronized (Godown[n]) {
         Sealed[n] += 1;
         if (packaged) {
            Godown[n] += 1;
            // Sealing complete
         }
      }
   }
}
```

If synchronization is not taken care of, updation of the data may be incorrect. For example in the above mentioned code of OrderManager, the number of packaged units in godown may not be updated correctly because of independent threads for Packaging Unit and Sealing Unit.

Example :

Say, at t=5, packaging unit completes packaging an already sealed bottle.

Also, at t=5, sealing unit completes sealing an already packaged bottle.

Thus, both the updates to the godown unit might be processed at the same time
Sealed Unit thread will process Godown[n] += 1.
Packaged Unit thread will process Godown[n] += 1.

Thus, if Godown[n] = 1 initially, after t=5, Godown[n] = 2.
But after synchronization, since both the threads can't update Godown[n] at the same time, one of them will update it before the other.
Hence, Godown[n] += 1 will give Godown[n] = 2, which will again be updated by other thread's Godown[n] += 1, which will result in final correct value Godown[n] = 3.

Concurrency:
Concurrency is achieved by multithreading. Individual threads are created for Packaging and Sealing and are being executed concurrently.

Synchronization:
I used block synchronization on accessing shared data, so that at a time only one thread can access the data.