Zad. 1

```
1  public class SimpleSnapshot<T> implements Snapshot<T> {
2    private StampedValue<T>[] a_table; // tablica atomowych rejestrów MRSW
3    public SimpleSnapshot(int capacity, T init) {
4      a_table = (StampedValue<T>[]) new StampedValue[capacity];
5      for (int i = 0; i < capacity; i++) {
6        a_table[i] = new StampedValue<T>(init);
7      }
8    }
9    public void update(T value) {
10     int me = ThreadID.get();
11     StampedValue<T> oldValue = a_table[me];
12     StampedValue<T> newValue =
13         new StampedValue<T>((oldValue.stamp)+1, value);
14     a_table[me] = newValue;
15   }
16   private StampedValue<T>[] collect() {
17     StampedValue<T>[] copy = (StampedValue<T>[])
18         new StampedValue[a_table.length];
19     for (int j = 0; j < a_table.length; j++)
20       copy[j] = a_table[j];
21     return copy;
22   }
23   public T[] scan() {
24     StampedValue<T>[] oldCopy, newCopy;
25     oldCopy = collect();
26     collect: while (true) {
27       newCopy = collect();
28       if (! Arrays.equals(oldCopy, newCopy)) {
29         oldCopy = newCopy;
30         continue collect;
31       }
32       T[] result = (T[]) new Object[a_table.length];
33       for (int j = 0; j < a_table.length; j++)
34         result[j] = newCopy[j].value;
35       return result;
36     }
37   }
38 }
```

1. Dowód poprawności
   Załóżmy nie wprost, że migawka nie działa prawidłowo i zwróciła błędny wynik.
   Aby do tego doszło, Arrays.equals(oldCopy, newCopy) == true. Może do tego dojść tylko wtedy, gdy nie doszło do zmiany żadnej sygnatury, co oznacza, że nie doszło do zmiany wartości w czasie kopiowania. Tym samym tablica jest poprawnym snapshotem. Sprzeczność.
2. Metoda scan:
   Może się zdarzyć tak, że wielokrotnie Array.equals(oldCopy, newCopy) == false. Będzie tak, gdy update() będziemy wykonywać na tyle często, że scan nie będzie w stanie wykonać 2 collectów między nimi. Zauważmy jednak, że gdy odizolujemy wątek scan(), to nie zmienią się wartości w tablicy, zatem zajdzie Array.equals(oldCopy, newCopy) == true. Scan będzie miał własność obstruction-free.
3. Metoda update:
   W metodzie nie ma pętli zatem wywołanie zawsze dokonuje postęp, co oznacza, że metoda jest lock-free.

Zad. 2

```
public class StampedSnap<T> {
  public long stamp;
  public T value;
  public T[] snap;
  public StampedSnap(T value) {
    stamp = 0;
    value = value;
    snap = null;
  }
  public StampedSnap(long ts, T v, T[] s) {
    stamp = ts;
    value = v;
    snap = s;
  }
}
```

```
public class WFSnapshot<T> implements Snapshot<T> {
  private StampedSnap<T>[] a_table; // array of atomic MRSW registers
  public WFSnapshot(int capacity, T init) {
    a_table = (StampedSnap<T>[]) new StampedSnap[capacity];
    for (int i = 0; i < a_table.length; i++) {
      a_table[i] = new StampedSnap<T>(init);
    }
  }
  private StampedSnap<T>[] collect() {
    StampedSnap<T>[] copy = (StampedSnap<T>[]) new StampedSnap[a_table.length];
    for (int j = 0; j < a_table.length; j++)
      copy[j] = a_table[j];
    return copy;
  }
  public void update(T value) {
    int me = ThreadID.get();
    T[] snap = scan();
    StampedSnap<T> oldValue = a_table[me];
    StampedSnap<T> newValue = new StampedSnap<T>(oldValue.stamp+1, value, snap);
    a_table[me] = newValue;
  }
  public T[] scan() {
    StampedSnap<T>[] oldCopy, newCopy;
    boolean[] moved = new boolean[a_table.length]; // initially all false
    oldCopy = collect();
    collect: while (true) {
      newCopy = collect();
      for (int j = 0; j < a_table.length; j++) {
        if (oldCopy[j].stamp != newCopy[j].stamp) {
          if (moved[j]) {
            return newCopy[j].snap;
          } else {
            moved[j] = true;
            oldCopy = newCopy;
            continue collect;
          }
        }
      }
      T[] result = (T[]) new Object[a_table.length];
      for (int j = 0; j < a_table.length; j++)
        result[j] = newCopy[j].value;
      return result;
    }
  }
}
```

Nieczekająca konstrukcja jest oparta na obserwacji, że jeśli skanujący wątek A widzi ruch wątku B dwa razy podczas powtórzonych collect, wtedy B wywołuje całkowicie update() w trakcie trwania scan() wątku A, więc jest poprawne dla A użycie snapshotu B.

Dowód poprawności:

Lemat 1. Jeśli wątek skanujący wykonuje czyste podwójne collect() to wartości, które zwraca istniały w rejestrach w pewnym stanie wykonania.
Dowód: Rozważ przedział między ostatnim odczytem pierwszego collect() a pierwszym odczytem drugiego collect(). Gdyby jakikolwiek rejestr został zaktualizowany w tym przedziale, znaczniki czasu nie pasowałyby, a podwójne collect() nie byłoby czyste.

Lemat 2. Jeśli wątek skanujący A obserwuje zmiany w znaczniku czasu innego wątku B podczas dwóch różnych podwójnych collect(), to wartość rejestru B odczytana podczas ostatniego collect() została zapisana przez wywołanie update(), które rozpoczęło się po rozpoczęciu pierwszego collect().
Dowód: Jeśli podczas scan (), dwa kolejne odczyty rejestru B przez A zwracają różne znaczniki czasu, to przynajmniej jeden zapis przez B następuje między tą parą odczytów. Wątek B zapisuje do swojego rejestru jako ostatni krok wywołania update(), więc pewne wywołanie update() przez B zakończyło się jakiś czas po pierwszym odczycie przez A, a krok zapisu kolejnego wywołania update() występuje między ostatnią parą odczytów przez A. Twierdzenie wynika z tego, że tylko B zapisuje do swojego rejestru.

Lemat 3. Wartości zwrócone przez scan() znajdowały się w rejestrach w pewnym czasie w trakcie wywołania metody.

Dowód: Jeśli wywołanie scan() wykonało czyste podwójne zbieranie, to twierdzenie wynika z Lematu 1. Jeśli wywołanie pobrało wartość skanowania z rejestru innego wątku B, to zgodnie z Lematem 2 wartość skanowania znaleziona w rejestrze B została uzyskana przez wywołanie scan() przez B, którego odstęp leży między pierwszym a ostatnim odczytem rejestru B przez A. Albo wywołanie funkcji scan() przez B miało czysty podwójny zbiór, w którym to przypadku wynik wynika z Lematu 1, albo istnieje osadzone wywołanie funkcji scan() przez wątek C występujące w przedziale wywołania funkcji scan() przez B. Ten argument można zastosować indukcyjnie, zauważając, że może być co najwyżej n − 1 zagnieżdżonych wywołań, zanim skończą się wątki, gdzie n jest maksymalną liczbą wątków. Ostatecznie, pewne zagnieżdżone wywołanie funkcji scan() musiało mieć czysty podwójny collect().

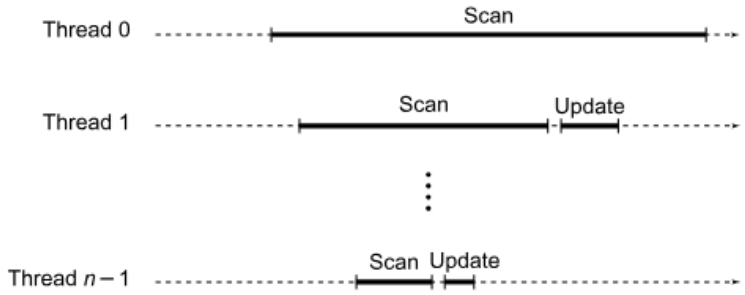


FIGURE 4.21

There can be at most $n - 1$ nested calls of `scan()` before we run out of threads, where n is the maximum number of threads. The `scan()` by thread $n - 1$, contained in the intervals of all other `scan()` calls, must have a clean double collect.

Lemat 4. Każde scan() lub update() zwraca po co najwyżej O(n2) odczytach lub zapisach.
Dowód: Rozważmy konkretne scan(). Istnieje conajwyżej n − 1 innych wątków, więc po n podwójnych collect(), albo jeden jest czysty, albo jakiś wątek porusza się dwa razy. Twierdzenie wynika, ponieważ każdy podwójny zbiór wykonuje O(n) odczytów.

Zad. 3

Załóżmy nie wprost, że istnieje nieczekająca implementacja protokołu binarnego konsensusu dla n wątków, używająca jedynie rejestrów atomowych.
Pokażemy, że przy takim założeniu istnieje nieczekająca implementacja protokołu binarnego konsensusu, używająca jedynie rejestrów atomowych dla n = 2.
Na początku mamy 2 wątki, które ustalają miedzy sobą wartość. Następnie uruchamiają się pozostałe n-2 i ich rozwiązaniem jest to, które ustaliły pierwsze 2.
Ale wiemy z wykładu, że implementacja dla dwóch wątków nie istnieje. Zatem nie mamy rozwiązania też dla n wątków. Sprzeczność.

Zad. 4

```
class GeneralConsProt<T> {
  private BinaryConsensus cons[];
  private AtomicRegister<T> propose[];

  public GeneralConsProt () {
    propose = new AtomicRegister<T>[N];
    cons = new BinaryConsensus[N];
  }

  public T decide(T value) {
    int me = ThreadID.get();
    propose[me] = value;
    cons[me].decide(true);

    for(int i = 0; i < N; i++)
       if(cons[i].decide(false))
          return propose[i];
  }
}
```

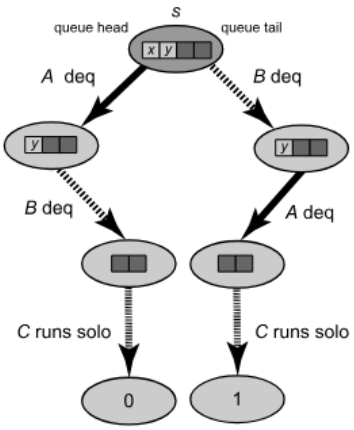
a) decide() zwraca jakąś wartość zaproponowaną przez wątki?
   Załóżmy, że żaden wątek nie ustawił konsensusu na true. Wtedy wszystkie musiały ustawić swojego poprzednika, ale aby do tego doszło poprzednik musiał ustawić swój na true. Sprzeczność.

b) Zwraca zawsze tą samą wartość?
   Weźmy dowolne wątki A i B oraz załóżmy, że zwracają inne wartości i BSO A < B.
   Żeby zwrócone wartości były inne musi zajść dla A cons[A].decide(false) == true,
   a dla B cons[A].decide(false) == false i cons[B].decide(false) == true.
   Sprzeczność.

Zad. 5

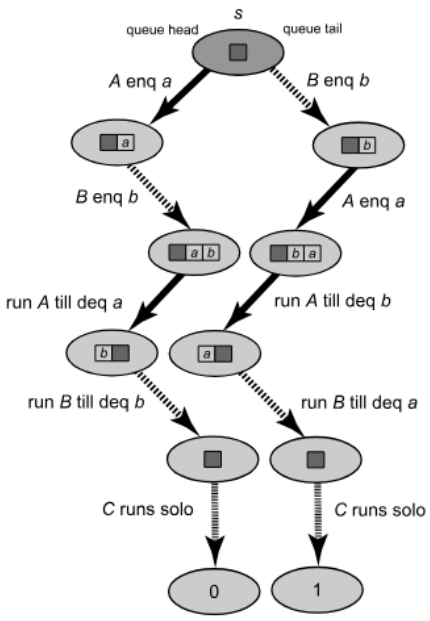
Załóżmy nie wprost, że da się taki konsensus osiągnąć dla wątków A, B, C. Rozpatrzymy możliwe scenariusze.

1. A: deq(), B: deq()



Niech S’ będzie stanem protokołu, kiedy A robi deq przed B i niech S’’ będzie stanem odwrotnym.
Wiemy, że S’ jest 0-valent, a S’’ 1-valent.
Jeśli C działa z S’ to decyduje 0, jeśli z S’’ to 1.
Jednak S’ i S’’ są nie rozróżnialne dla C (te same elementy usunięte z kolejki), zatem C musi zdecydować w obu stanach. Sprzeczność.

2. A: enq(a), B: enq(b)



Niech S' to stan po wywołaniu:
Wykonujemy A dopóki nie ściągnie z kolejki a.
(Jedyny sposób na obserwacje kolejki to deq(), więc A nie może decydować, dopóki nie zobaczy a lub b)
Zanim A wykona się dalej, uruchamiamy B dopóki nie ściągnie z kolejki b.
Niech S'' to stan po wywołaniu:
B i A dodaje do kolejki elementy b i a, w danej kolejności. Uruchamiamy A dopóki nie ściągnie b.
Zanim A wykona się dalej, uruchamiamy B dopóki nie ściągnie z kolejki a.
S' jest 0-valent, a S'' jest 1-valent.
Oba wywołania A są identyczne, dopóki nie zwróci a lub b. Skoro A jest zatrzymane zanim zmieni cokolwiek, to B też jest identyczne do momentu zwrócenia a lub b. Wiemy, że A i B są nierozróżnialne przez C. Sprzeczność.

3. A: enq(a), B: deq()
   Jeżeli kolejka nie jest pusta to od razu mamy sprzeczność, ponieważ obie metody działają na różnych końcach kolejki, więc C nie może rozróżnić w jakiej kolejności je wprowadzono.
   Jeśli kolejka jest pusta, wtedy osiągany stan 1-valent (B robi deq() na pustej kolejce, później wykonuje się A) jest dla C nierozróżnialny od 0-valent stanu (A samo wywołuje enq(a))
   (jeśli po A wykona się B to kolejka będzie ostatecznie pusta, więc C nie będzie mogło zobaczyć czy wykonały się te operacje)

Zad. 6

a) Wartość zwracana jest jedną z zaproponowanych przez wątki.
   Załóżmy nie wprost, że zwrócono wartość różną od zaproponowanych przez dwa wątki.
   Zauważmy, że aby do tego doszło, wątek pierwszy musiałby wykonać return proposed[j] na niezainicjowanej proposed[j]. Dochodzimy do sprzeczności, ponieważ wtedy musiało zajść position[i] < position[j], co implikuje przypisanie wartości proposed[j]

b) Zwraca tą samą wartość obydwu wątkom.
   Załóżmy nie wprost, że wątki zwróciły różne wartości.

   else if (position[i] < position[j]) // I am behind you
   return proposed[j];
   Jeden z wątków musiał wykonać to jako pierwszy. Wiemy, że wartości postion, mogą tylko rosnąć, zatem niemożliwe jest, aby drugi wątek odczytał swoje position jako mniejsze.

   if (position[i] > position[j] + speed[j]) // I am far ahead of you
   return proposed[i];
   Jeden z wątków musiał wykonać to jako pierwszy, zatem drugi z wątków na pewno wykona drugiego ifa i zwróci proposed[j].

Poziom konsensusu odnosi się do algorytmów wait-free, jednak rozwiązanie to nie jest wait-free. W przypadku kiedy wartości position są identyczne, wolniejszy wątek może wykonać 3 razy pętle while, kiedy szybszy wykona ją raz, co wróci nas do punktu wyjścia (nieskończona pętla).

Zad. 7

1. Każdy wątek zapisuje swoją wartość. StickyBIt pamięta jedynie pierwszą zapisaną wartość. Wątki odczytujące zawsze będą zgodne, bo będą czytały to samo.

2. Niech i-ty bit wynik[i] będzie obiektem klasy StickyBit zapisany w rejestrze.
   Każdy wątek dostanie swój rejestr MRSW (R).

   Algorytm:

   1. Dla każdego wątku R[i] = wartości i-tego wątku
   2. Dla każdego wątku ID = 0 do N:
      a. Dla każdego i = 0 do log_m:
         i. Jeśli wynik[i] == ⊥ lub wynik[i] == R[ID][i]: wynik[i] = R[ID][i]
         ii. wpp: dla każdego id2=0 do n:
            1. jeśli wynik[0..i]==R[id2][0…i]: R[ID] = R[id2]

Każdy wątek zwróci te samą wartość.

Zad. 8

Pokażemy implementację protokołu przybliżonej zgody używającej jedynie rejestrów atomowych.

1. Ustaw rejestry na -1
2. Każdy wątek zapisuje swoją wartość do swojego rejestru
3. Jeśli drugi rejestr ma wciąż wartość -1, to zwracamy swoją wartość
4. Każdy wątek aktualizuje swoją wartość:
   4.1. Jeśli |ya - yb| <= epsilon, to zwracamy swoją wartość
   4.2. Wpp. zapisujemy |ya - yb|/2 do danego rejestru i wracamy do kroku 4

Protokół jest wtedy wait-free, bo każdy wątek działa niezależnie od drugiego. Zatem poziom konsensusu obiektów przybliżonej zgody jest równy 1.