Zad. 1

Lokalność czasowa:
Jeśli program korzysta z komórki pamięci, w niedługim czasie ponownie z niej skorzysta.

Lokalność miejscowa:
Jeśli program korzysta z komórki pamięci, w niedługim czasie skorzysta z komórki blisko niej.

Ładując komórkę do pamięci podręcznej, przenosi się tam wiersz bliskich sobie komórek.

Spójność pamięci podręcznych to spójność danych dzielonych przez różne pamięci podręczne. Gdyby spójność nie zachodziła, różne procesory, odnoszące się do jednej komórki pamięci, mogłyby odczytywać już nieaktualne dane (inny procesor zmodyfikował komórkę, ale reszta procesorów nie zaktualizowała pamięci podręcznych). Protokoły spójności temu zapobiegają.

MESI to prosty protokół spójności, działający na 4 stanach pamięci podręcznej:

1. **M**odified - procesor zmodyfikował komórkę pamięci podręcznej i musi zapisać ją do pamięci głównej.
2. **E**xclusive - wiersz pamięci podręcznej posiada te same dane, co pamięć główna, dodatkowo wyłącznie on posiada ten wiersz w swojej pamięci podręcznej.
3. **S**hared - jak wyżej, ale procesor może dzielić ten sam wiersz z innym procesorem.
4. **I**nvalid - wiersz pamięci podręcznej posiada nieaktualne dane.

Gdy procesor rząda dostępu do pamięci głównej, a żaden inny nie ma go we własnej pamięci, otrzyma go w stanie Exclusive. Gdyby inny procesor domagał się tego samego wiersza pamięci, nie otrzyma go od pamięci głównej, ale z pamięci odpowiedniego procesora, a wiersze zyskują stan Shared.
By ograniczyć wykorzystanie pamięci, gdy procesor modyfikuje wartość komórki swojej pamięci podręcznej, wiersz zyskuje stan Modified, a procesory, które go dzieliły, zmieniają własny wiersz na Invalid. Pamięć główna zapisze dane dopiero wtedy, gdy procesor opróżni wiersz z własnej pamięci (Write-Back).

Zad. 2

Test-and-set Lock

```
class TASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

 void lock() {
  while (state.getAndSet(true)) {}
 }

 void unlock() {
  state.set(false);
 }}
```

Test-and-test-and-set Lock

```
class TTASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

 void lock() {
  while (true) {
   while (state.get()) {}
   if (!state.getAndSet(true))
    return;
 }
}
```

1. Układa się prosty program z krótką sekcją krytyczną (np. licznik) i mierzy się czas wykonania programu i średni czas potrzeby na zajęcie zamka
2.
    a. Zamek TAS przez próbę zapisu za każdym razem unieważnia pamięć podręczną we wszystkich procesach, przez co musi być aktualizowana z pamięci współdzielonej.
    b. Zamek TTAS tylko odczytuje stan, przez co wszystkie pamięci nie są unieważniane gdy nie ma możliwości zablokowania zamka, tylko wtedy gdy się to udaje.

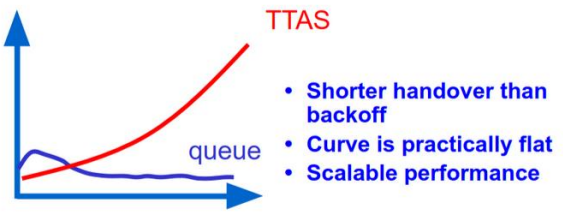
Zad. 3

Anderson Queue Lock

```
public lock() {
 mySlot = next.getAndIncrement();
 while (!flags[mySlot % n]) {};
 flags[mySlot % n] = false;
}

public unlock() {
 flags[(mySlot+1) % n] = true;
}
```

Performance



Fałszywe współdzielenie występuje, gdy różne wątki modyfikują dane przechowywane w tej samej linii pamięci podręcznej. Może to prowadzić do dużej liczby niepotrzebnych invalidacji w cache, spowalniając wydajność.

Aby rozwiązać ten problem między elementami tablicy flags należy dodać „padding", w taki sposób, że każdemu wierszowi pamięci przypada dokładnie jedna ważna komórka flags. Teraz przy modyfikacjach, invalidacja wpłynie tylko na jedną komórke zamiast na wszystkie mieszczące się w linii.

Zad. 4

1. Każdy wątek modyfikuje wspólną zmienną chronioną zamkiem. Duża liczba odwołań do tej samej pamięci podręcznej powoduje konflikt i zwiększa czas oczekiwania.
2. Podejście podobne do Andersona. Każdy wątek modyfikuje inną wartość w tablicy, co zmniejsza ryzyko konfliktów w pamięci podręcznej. Stosując padding, kosztem większego zużycia pamięci zyskamy na czasie na dodatkowe operacje na szynie danych, a więc uzyskamy większe przyspieszenie.

Zad. 5

CLH Queue Lock

```
class CLHLock implements Lock {
 AtomicReference<QNode> tail;
 ThreadLocal<QNode> myNode
    = new QNode();
 public void lock() {
  QNode pred
    = tail.getAndSet(myNode);
  while (pred.locked) {}
 }}
```

Zamek CLH działa podobnie do zamka Andersona, ale zamiast tablicy komórek mamy niejawną listę wiązaną. W momencie, kiedy wątek chce przejść do sekcji krytycznej, alokuje nowy węzeł z domyślną wartością true oraz zamienia się wskaźnikami z ogonem. Dopóki pred nie zmieni swojej wartości na false, wątek czeka w pętli. Kiedy sekcja krytyczna zostaje zwolniona, węzeł wskazywany przez pred zmienia swoją wartość na false i czekający wątek może wejść do sekcji krytycznej.

Wyżej zostało wspomniane, że obiekt, mówiący o stanie wątku czekającego w kolejce, jest tworzony w momencie zgłoszenia chęci nabycia zamka. Można jednak ograniczyć ilość takich alokacji poprzez przydzielenie danemu wątkowi przy ponownej próbie nabycia zamka, obiektu węzła wcześniej wykorzystanego przez ten wątek. Gdy wątek zwalnia zamek, jego obiekt poprzednika nie jest już nikomu potrzebny i może zostać w ten sposób użyty ponownie.

Zad. 6

Zadanie 6. Poniżej znajduje się alternatywna implementacja zamka CLHLock, w której wątek ponownie wykorzystuje nie węzeł swojego poprzednika, ale własny. Wyjaśnij, dlaczego ta implementacja jest błędna.

```
public class BadCLHLock implements Lock {
    AtomicReference<Qnode> tail = new AtomicReference<QNode>(new QNode());
    ThreadLocal<Qnode> myNode = new ThreadLocal<QNode> {
        protected QNode initialValue() {
            return new QNode();
        }
    };
    public void lock() {
        Qnode qnode = myNode.get();
        qnode.locked = true; // I'm not done
        // Make me the new tail, and find my predecessor
        Qnode pred = tail.getAndSet(qnode);
        while (pred.locked) {}
    }
    public void unlock() {
        // reuse my node next time
        myNode.get().locked = false;
    }
    static class Qnode { // Queue node inner class
        volatile boolean locked = false;
    }
}
```

Rozpatrzmy następujący scenariusz jednowątkowy:

1. Wątek wykonuje lock: ustawiamy tail == myNode.get(), myNode.get().locked == true
2. Wątek wykonuje unlock: ustawiamy tail == myNode.get(), myNode.get().locked == false
3. Wątek ponownie wykonuje lock:
   a) qnode.locked = true: tail == myNode.get(), myNode.get().locked == false
   b) qnode pred = tail.getAndSet(qnode): pred == myNode.get(), myNode.get().locked == true
   c) wątek wchodzi do while i utyka w pętli, ponieważ pred.locked == true.

Jak widać wątek pomimo bycia jedynym działającym utyka w pętli, a to pokazuje, że podana implementacja jest błędna.

Zad. 7

```
1  public class TOLock implements Lock{
2    static QNode AVAILABLE = new QNode();
3    AtomicReference<QNode> tail;
4    ThreadLocal<QNode> myNode;
5    public TOLock() {
6      tail = new AtomicReference<QNode>(null);
7      myNode = new ThreadLocal<QNode>() {
8        protected QNode initialValue() {
9          return new QNode();
10       }
11     };
12   }
13   ...
14   static class QNode {
15     public volatile QNode pred = null;
16   }
17 }
```

FIGURE 7.13

TOLock class: fields, constructor, and QNode class.

```
18 public boolean tryLock(long time, TimeUnit unit) throws InterruptedException {
19   long startTime = System.currentTimeMillis();
20   long patience = TimeUnit.MILLISECONDS.convert(time, unit);
21   QNode qnode = new QNode();
22   myNode.set(qnode);
23   qnode.pred = null;
24   QNode myPred = tail.getAndSet(qnode);
25   if (myPred == null || myPred.pred == AVAILABLE) {
26     return true;
27   }
28   while (System.currentTimeMillis() - startTime < patience) {
29     QNode predPred = myPred.pred;
30     if (predPred == AVAILABLE) {
31       return true;
32     } else if (predPred != null) {
33       myPred = predPred;
34     }
35   }
36   if (!tail.compareAndSet(qnode, myPred))
37     qnode.pred = myPred;
38   return false;
39 }
40 public void unlock() {
41   QNode qnode = myNode.get();
42   if (!tail.compareAndSet(qnode, null))
43     qnode.pred = AVAILABLE;
44 }
```

FIGURE 7.14

TOLock class: tryLock() and unlock() methods.

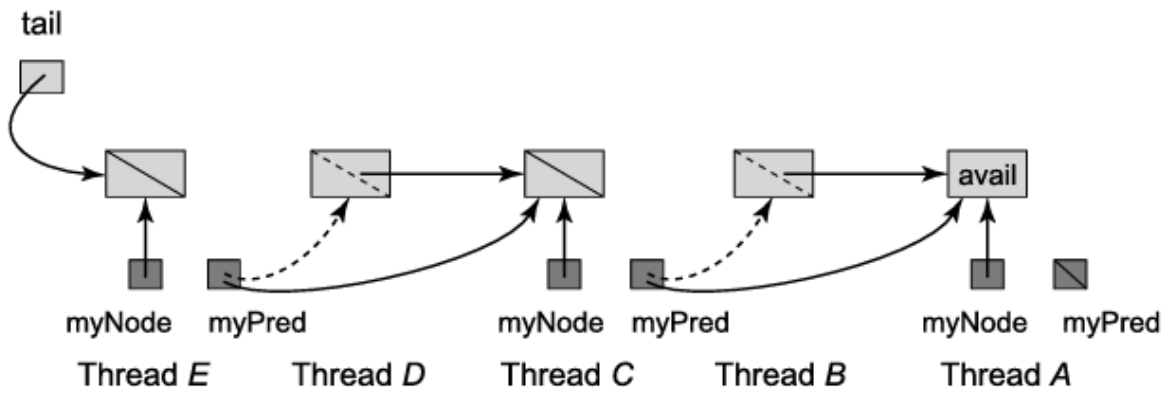


FIGURE 7.15

Timed-out nodes that must be skipped to acquire the TOLock. Threads *B* and *D* have timed out, redirecting their pred fields to their predecessors in the list. Thread *C* notices that *B*'s field is directed at *A* and so it starts spinning on *A*. Similarly, thread *E* spins waiting for *C*. When *A* completes and sets its pred to AVAILABLE, *C* will access the critical section and upon leaving it will set its pred to AVAILABLE, releasing *E*.

Zamek CLH z czasem ważności posiada metodę tryLock, dzięki której wątek może określić ile czasu jest w stanie poświęcić na zajmowanie zamka. Jeżeli wątek nie zajmie zamka w określonym czasie porzuca tę próbę. Próba zajęcia zamka kończy się zwróceniem zmiennej boolowskiej.
W klasycznym CLH przerwanie wykonywania metody lock przez wątek skutkowałoby zagłodzeniem wątków czekających za nim w kolejce. Metoda tryLock rozwiązuje ten problem przez oznaczenie wątku, który porzucił wykonywanie metody. Dzięki temu pozostałe wątki czekające w kolejce mogą wejść do sekcji krytycznej.

Zad. 9

a) TAS:

```
public boolean isLocked(){
        return state.get();
}
```

b) CLH:

```
public boolean isLocked(){
        return tail.get().locked;
}
```

c) MCS:

```
public boolean isLocked(){
return tail.get() != null;
}
```

Zad. 10

Mamy wielopoziomową hierarchię pamięci podzieloną na clustery zwierające w sobie procesory. Np. dwupoziomową i do pierwszego poziomu odwołujemy się szybko, a operacje na drugim poziomie są bardziej kosztowne. W takim przypadku warto zastąpić BOLock(Back-off Lock) poprzez HBOLock.

HBOLock to rozszrzenie idei Back-off Lock ktora polegała na tym, aby wątek robił coraz dłuższego sleepa w pętli w funkcji lock. Dzięki temu wątki oczekujące na dany zasób nie będą spamować zpytaniami o stan zmiennej i zmniejszy się użycie procesora.

Rozszerzenie w HBOLock polega na uwzględnieniu tego, że odpytanie o stan zmiennej niesie za sobą różny koszt czasowy w zalezności od tego na którym poziomie hierarchi była ta zmienna. Chcemy, aby kosztownych zapytań było jak najmniej. HBOLock z tego powodu będzie zwiększać czas sleepa w wątkach z innego clusteru bardziej niż w wątkach w obrębie tego samego clusteru.

```
1  public class HBOLock implements Lock {
2    private static final int LOCAL_MIN_DELAY = ...;
3    private static final int LOCAL_MAX_DELAY = ...;
4    private static final int REMOTE_MIN_DELAY = ...;
5    private static final int REMOTE_MAX_DELAY = ...;
6    private static final int FREE = -1;
7    AtomicInteger state;
8    public HBOLock() {
9      state = new AtomicInteger(FREE);
10   }
11   public void lock() {
12     int myCluster = ThreadID.getCluster();
13     Backoff localBackoff =
14       new Backoff(LOCAL_MIN_DELAY, LOCAL_MAX_DELAY);
15     Backoff remoteBackoff =
16       new Backoff(REMOTE_MIN_DELAY, REMOTE_MAX_DELAY);
17     while (true) {
18       if (state.compareAndSet(FREE, myCluster)) {
19         return;
20       }
21       int lockState = state.get();
22       if (lockState == myCluster) {
23         localBackoff.backoff();
24       } else {
25         remoteBackoff.backoff();
26       }
27     }
28   }
29   public void unlock() {
30     state.set(FREE);
31   }
32 }
```

FIGURE 7.17
The HBOLock class: a hierarchical back-off lock.