Αναστασάς Αναστάσιος Α.Μ:3166
Login: csd3166

Exercise A

1. Θα χρησιμοποιήσουμε locks βασισμένα στο CLH-Lock μοντέλο όπως έχει περιγραφεί και στις διαφάνειες.

```
#define cartReady = 0;
#define startTour = 1;

shared int state = cartReady;        //keeping the state for the cart
shared boolean pass = false;         //true when a passenger come and it's ready
shared boolean cart = false;         //true when a cart come and it's ready

CLH_Lock currPassenger;              //CLH lock starvation-free
CLH_Lock currCart;                   //CLH lock starvation-free

void passenger(){
      walk_around();                 //do something before take the cart

      lock(currPassenger);           //try to lock, waiting until it's our state
      while(cart == false) noop;     //wait for cart
      pass = true;                   //declare passenger purpose
      while(state != startTour) noop; //spinning until the cart change on
                                          startTour
      pass = false;                  //declare that we dont need any more cart
      unlock(currPassenger);         //i am get a cart, give order on next passenger

      buy_something_from_shop();     //do something
}

void cart(){

      lock(currCart);                //try to lock, the first cart that lock will take
                                     the first passenger*/
      state = cartReady;             //declare the current state
      cart = true;                   //declare our purpose
      while(pass == false) noop;     //waiting passenger to come
      cart = false;                  //declare that our cart take by someone
      state = startTour;             //change state on startTour
      while(pass == true) noop;      //wait the passenger to join us
      unlock(currCart);              //leaving critical region

      start_tour();                  //start the tour
}
```

Αφού θα βασιστούμε στο CLH lock για το οποίο έχουμε αποδείξει ότι είναι starvation free συνεπώς θα είναι και deadlock free.

2. Έχουμε 2 συναρτήσεις μια για τα καροτσάκια και μια για τους επισκέπτες. Υποθέτουμε ότι το πολύ m διεργασίες θα τρέξουν τον αλγόριθμο cart().
   - void cart() : όποια διεργασία καταφέρει να κάνει lock θα πάρει τον $1^{ο}$ επισκέπτη και όσες έρθουν στην συνέχεια θα πάρουν μια σειρά εκτέλεσης μετά το τέλος εκτέλεσης τις $1^{η}$ διεργασίας. Το cart που θα μπει στην κρίσιμη περιοχή θα αλλάξει την κατάσταση του σε cartReady και θα δηλώσει ότι υπάρχει ένα καροτσάκι ώστε να ενημερωθεί και ο πελάτης. Στην συνέχεια περιμένουμε μέχρι να έρθει κάποιο πελάτης. Αφού έρθει ο πελάτης δηλώνει το cart ότι είναι μη διαθέσιμο και αλλάζουμε την κατάσταση του σε startTour. Τέλος περιμένουμε μέχρι να δηλώσει ο πελάτης ότι έφυγε ώστε να βγούμε από την κρίσιμη περιοχή.

- Void passenger(): αφού κάνει μια βόλτα ο επισκέπτης προσπαθεί να πάρει θέση για να κάνει βόλτα με το καροτσάκι, εάν είναι η σειρά του και καταφέρει να κάνει lock περιμένει μέχρι να υπάρξει διαθέσιμο cart. Ορίζει μετά ο επισκέπτης την πρόθεση του ότι θέλει καροτσάκι. Μετά κάνει spinning μέχρι να είναι έτοιμο και το cart για να αρχίσει το tour, ο πελάτης μετά δηλώνει ότι βρήκε cart άρα δεν είναι πλέον διαθέσιμος και βγαίνουμε από την κρίσιμη περιοχή.

3. Παρατηρούμε ότι ο αλγόριθμος cart εξαρτάται από τον αλγόριθμο του passenger. Επομένως όποια από τις δυο συναρτήσεις τρέξει πρώτη δεν έχει διαφορά στο αποτέλεσμα. Άρα τα σημεία σειριοποίησης μπορούν να μπουν σε οποιοδήποτε σημείο.

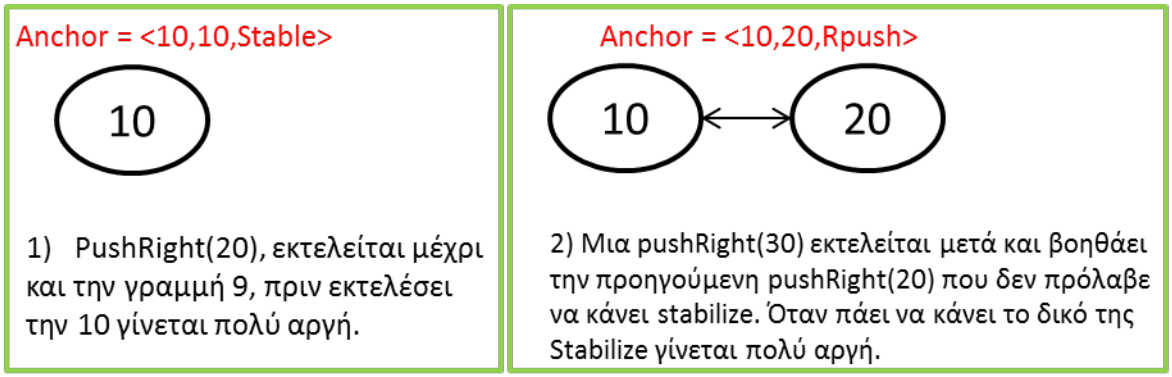
4. Ο τρόπος που θα υλοποιηθούν τα lock είναι starvation free επομένως είναι και deadlock free αφού είναι υποσύνολο του starvation. Μόνο 2 διεργασίες θα είναι στην κρίσιμη περιοχή, μια διεργασία cart & μια passenger. Και στις δυο συναρτήσεις υπάρχουν 2 while που κάνουν spinning, όπου η εκάστοτε διεργασία ξεμπλοκάρει μόνο από την άλλη διεργασία. Επόμενος η cart συνάρτηση κάνει progress με βάση το progress της passenger και το αντίστροφο.

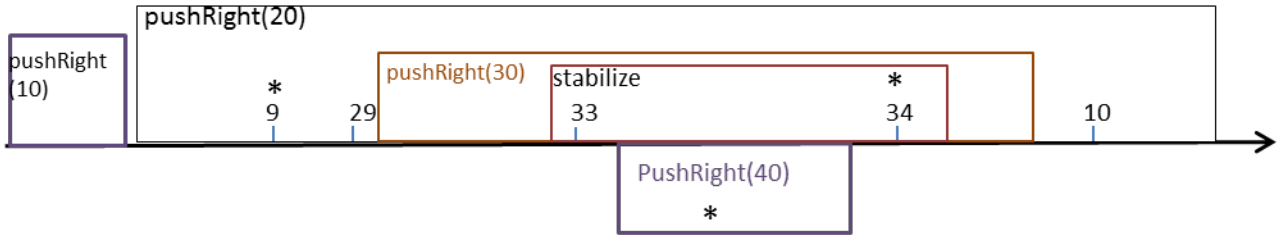
Αναστασάς Αναστάσιος Α.Μ:3166
Login: csd3166

Exercise B

1. Πιστεύω είναι σωστά τα lineazarization points γιατί στα σημεία που έχουν τοποθετηθεί είναι τα σημεία του κώδικα όπου η τρέχουσα διεργασία κάνει εμφανές στις άλλες διεργασίες κάποια νέα δεδομένα. Επομένως εάν βάζαμε πιο πριν τα σημεία θα ήταν πολύ νωρίς και αυτό θα μας οδηγούσε σε κακά σενάρια, το ίδιο εάν έμπαιναν λίγο αργότερα.

2.





Όπως φαίνεται και στο σχήμα παραπάνω η παράλληλη εκτέλεση θα μας έφτιαχνε μια στοίβα με τα στοιχεία 10,20,30,40
Στην σειριακή εκτέλεση όμως θα είχαμε μια στοίβα με τα στοιχεία 10,20,40,30 αφού με βάσει τα linearization σημεία που φαίνονται παραπάνω θα εκτελούνταν οι συναρτήσεις ως εξής push(10), push(20), push(40), push(30)
Άρα παρατηρούμε ότι εάν σειριοποιήσουμε στην 34 γραμμή όσες καταφέρουν να την εκτελέσουν την συγκεκριμένη CAS τότε οδηγούμαστε στο παραπάνω σενάριο.

3. Με βάση την μελέτη που κάναμε στο ερώτημα 1 και 2 πιστεύω ότι τα σημεία σειριοποίησης πρέπει να τοποθετηθούν όπως τοποθετήθηκαν και στην άσκηση 1. Η συνάρτηση stabilize απλά βοηθάει κάποια μη ολοκληρωμένη push,pop να εκτελεστεί. Επομένως το να βάλουμε μέσα σε αυτήν κάποιο σημείο σειριοποίησης θα μας οδηγούσε σε κακά σενάριο όπως στον 2^{ο} ερώτημα. Στην pushRight εάν βάζαμε σημεία σειριοποίησης στις γραμμές 13,14 δεν θα ήταν και πάλι σωστά γιατί σε εκείνα τα σημεία απλά ενώνει

κάποιους pointer από προηγούμενες pop,push που δεν ολοκληρώθηκα, επομένως δεν βάζει κάτι νέο στο σύστημα αλλά το ολοκληρώνει επειδή είχε μείνει ημιτελές. Το ίδιο ισχύει και για την γραμμή 10. Στην γραμμή 8

απλά αλλάζουμε έναν δείκτη μιας τοπικής μεταβλητής και η οποίο δεν γίνεται εμφανής σε αυτό το σημείο, άρα και εδώ θα ήταν πολύ νωρίς ένα σημείο σειριοποίησης. Για του ίδιους λόγους δεν μπορεί και στην pop να μπει σε κάποιο διαφορετικό σημείο τα linearize points.
Επομένως τα linearize points είναι τα παρακάτω:

- pushRight/pushLeft:
  - 6. if (CAS (&Anchor,<L,R,STABLE> , <node,node,RPUSH))return;
  - 9. If(CAS(&Anchor, <L,R,STABLE>,<L,node,RPUSH)) {
- popRight/popLeft:
  - 17. If(R== NULL) return NULL;
  - 19. If(CAS(&Anchor, <L,R,STABLE>, <NULL,NULL,STABLE>)) break;
  - 22. if(CAS(&Anchor, <L,R,STABLE>, <L,prev,STABLE>)) break;

4. Όχι δεν είναι wait-free ο αλγόριθμος γιατί έστω το εξής σενάριο:
   Έστω ότι έχει γίνει ένα pushRight αλλά για κάποιον λόγω δεν έχει μπρολάβει να κάνει το stabilize τότε έρχεται μια popRight η οποία βοηθάει το stabilize για την προηγούμενη pushRight. Μόλις εκτελεστεί η γραμμή 29 του stabilize γίνεται πάρα πολύ αργή και έρχεται μια άλλη pushRight η οποία κάνει το stabilize και προσπαθεί να κάνει push το δικό της στοιχείο. Πριν όμως κάνει αυτή το stabilize γίνεται και αυτή πολύ αργή. Συνεχίζει η προηγούμενη pop η οποίο στην γραμμή 30 βρίσκει true την συνθήκη και επιστρέφει, και πάει να κάνει πάλι stabilize και ξανα γίνεται αργή στην γραμμή 29 όπως και πριν και έρχεται πάλι μια άλλη push και κανει stabilize το προηγούμενο στοιχείο, αλλάζει το anchor όπως και πριν και δεν κάνει πάλι Stabilize….. αυτό επαναλαμβάνεται διαρκώς και έτσι η pop δεν θα τελειώσει ποτέ.
5. Εάν αφαιρέσουμε αυτές τις γραμμές τότε αν υποθέσουμε ότι μια pushRight ξεκινήσει και τρέχει και αλλάζει το anchor status σε rpush και πριν πάει να τρέξει την stabilize crasharei και δεν έρχεται καμία άλλη push ώστε να την βοηθήσει. Τότε εάν τρέξουμε μια popRight αυτή θα τρέχει για πάντα καθώς το state δεν θα αλλάξει ποτέ σε Stable.
6. Έστω ότι έχουμε έναν register R/W ο οποίος μας παρέχει δύο λειτουργίες, Read και Write. Εάν υποθέσουμε ότι αυτός ο καταχωρητής κρατάει ένα διάνυσμα των δεδομένων. Τότε εάν διαβάσουμε με την Read (ατομικά) αυτό το διάνυσμα από τον καταχωρητή και στην συνέχεια σπάσουμε αυτό το διάνυσμα στα 3 δεδομένα μας άρα μπορούμε να πετύχουμε την ίδια λειτουργικότητα απλά θα είναι πιο αργός ο αλγόριθμος μας.

7. –

Αναστασάς Αναστάσιος Α.Μ:3166
Login: csd3166

Exercise C

A. Coarse-grained sync
   1. Ψευδοκώδικας

```
1) #define EMPTY 1
2) #define WAITING 2
3) #define BUSY 3
4) #define CAPACITY 100
5)
6) //Each element of exchanger[] is initially <null, EMPTY>.
7) <T, state> exchanger[CAPACITY];
8) state = {EMPTY, WAITING, BUSY}
9) //Locks implemented with mutex lock
10) shared Lock exchange;
11) shared Lock stack;
12)
13) T visit(T value, int range, long duration) {
14)     int el = randomnumber(range);
15)     return (exchange(exchanger[el], value, duration));
16) }
17)
18)
19)T exchange(shared <T, int> slot, T myitem, long timeout) {
20)    long timeBound = getnanos() + timeout;
21)    while (TRUE) {
22)      // if it is time for timeout, leave the exchanger
23)      if (getnanos() > timeBound) return TIMEOUT;
24)      lock(exchange);
25)      <youritem, state> = slot;
26)      switch (state) {
27)          case EMPTY: // try to place your item in the slot and set state to WAITING
28)              slot = <myitem, WAITING>;
29)              unlock(exchange);
30)              while (getnanos() < TimeBound) { // spin until it is time for timeout
31)                  <youritem, state> = slot; // read slot
32)                  if (state == BUSY) {
33)                      slot = <null, EMPTY>; // if the exchange is complete
34)                      return youritem;
35)                  } // return the other process's item
36)              }
37)              // if no other thread shows up
38)              if (slot == myitem && state == WAITING) { //linearization *
39)                  slot = <null, EMPTY>
40)                  return TIMEOUT
41)              } else {
42)                  lock(exchange);
43)                  <youritem, state> = slot;
44)                  slot = <null, EMPTY>;
45)                  unlock(exchange);
46)                  return youritem;
47)              }
48)          case WAITING: // some thread is waiting and slot contains its item
49)              slot = <myitem, BUSY>; //linearization *
50)              unlock(exchange);
51)              return youritem; // and return the item of the other process
52)          case BUSY: // two other threads are currently using the slot
                 unlock(exchange);
53)              break; // the process must retry
54)      } // switch
55) }
56)}
```

```
57)void push(T x) {
58)    int range;
59)    long duration;
60)    NODE *nd = newcell(NODE);
61)    nd->value = x;
62)    // try to use the elimination array, instead of backing-off
63)    range = CalculateRange(); // choose the range parameter
64)    duration = CalculateDuration(); // choose the duration parameter
65)    otherValue = visit(x, range, duration); // call visit with input value as argument
66)    if (otherValue == NULL) {//check whether the value was exchanged with a pop()method
67)           RecordSuccess(); // if yes, record success
68)           return;
69)    } else if (otherValue == TIMEOUT) // otherwise,
70)           RecordFailure(); // record failure
71)    lock(stack);
72)    NODE * oldTop = TOP;
73)    nd->next = oldTop;
74)    Top = nd; //linearization point *
75)    unlock(stack);
76)}
```

```
77)T pop(void) {
78)    range = CalculateRange(); // choose the range parameter
79)    duration = CalculateDuration(); // choose the duration parameter
80)    otherValue = visit(NULL, range, duration);//call visit with input value as argument
81)    if (otherValue != NULL) {//check whether the value was exchanged with a push()method
82)           RecordSuccess(); // if yes, record success
83)           return otherValue;
84)    } else if (otherValue == TIMEOUT) // otherwise,
85)           RecordFailure(); // record failure
86)    lock(stack);
87)    NODE *oldTop = Top;
88)    if (oldTop == NULL) {
89)           unlock(stackLock);
90)           return EMPTY_STACK;
91)    }
92)    Top = oldTop->next; //linearization *
93)    unlock(stack);
94)}
```

2. Ο αλγόριθμος είναι βασισμένος στον lock-free exchanger. Αρχικά η push πηγαίνει στον elimination array και ελέγχει αν υπάρχει κάποια άλλη διεργασία στο συγκεκριμένο slot του πίνακα. Εάν δεν καταφέρει να γίνει κάποιο ραντεβού στον πίνακα τότε το στοιχείο γίνεται push όταν καταφέρει να κάνει lock την στοίβα ολόκληρη. Παρόμοια ισχύει και για την pop.
   Η exchange θα κάνει τον έλεγχο για elimination. Κάνουμε lock ώστε να διαβάσουμε εμείς το slot και να εξασφαλίσουμε ότι δεν θα αλλάξει μέχρι να το διαβάσουμε και να κάνουμε τους ελέγχους που πρέπει. Αν είναι άδειο τότε πρέπει να μπούμε εμείς και να περιμένουμε μέχρι να έρθει κάποια άλλη διεργασία ή συμβεί timeout. Όταν το slot είναι σε state waiting τότε το κάνουμε busy και επιστρέφουμε την τιμή που είχε μέσα το slot. Γενικά κάνουμε lock κάθε φορά που πάμε να διαβάσουμε από μια shared θέσει όπως είναι το slot ενός πίνακα. Και κάνουμε unlock όταν δεν θα διαβάσουμε η γράψουμε άμεσα κάτι σε κοινόχρηστη μεταβλητή η οποία θα μπορούσε να δημιουργήσει κακά σενάρια.

```
while (getnanos() < TimeBound) { // spin until it is time for timeo
                <youritem, state> = slot; // read slot
```

   στο παραπάνω κομμάτι κώδικα δεν είναι αναγκαίο το lock καθώς κάνουμε spinning άρα εάν διαβάσει κάτι την στιγμή που αλλάζει τότε στην επόμενη ανακύκλωση θα το διαβάσουμε σωστά.
   Θεωρούμε ότι το διάβασμα γίνεται ατομικά.

3.

Για την exchange

- Εάν πετύχει η exchange τότε σειριοποιούμε και τις 2 διεργασίες που συναντήθηκαν όταν η $2^{η}$ διεργασία αλλάξει την κατάσταση του slot από waiting σε busy. Αρά στην γραμμή 49.
- Εάν δεν πετύχει η exchange την σειριοποιούμε στην γραμμή 38.

Για την push

- Εάν πετύχει το elimination τότε την σειριοποιούμε όπου είπαμε για την exchange
- Αλλιώς στην 74 γραμμή όπου αλλάζει το top και έτσι γίνεται εμφανές η αλλαγή στους άλλους.

Για την Pop

- Εάν πετύχει το elimination τότε την σειριοποιούμε όπου είπαμε για την exchange
- Αλλιώς στην 92 γραμμή όπου αλλάζει το top και έτσι γίνεται εμφανές η αλλαγή στους άλλους.

4. Υποθέτουμε ότι τα linearization points είναι αυτά που δώσαμε στο ερώτημα 3. Εάν υποθέσουμε ότι είναι σωστά τότε ο αλγόριθμος μας είναι serializable. Επομένως είναι και σωστός.

5. Ο αλγόριθμος μας θα έχει συνέχεια progress γιατί ο αλγόριθμος είναι starvation free. Ο λόγος που είναι starvation free είναι επειδή κάθε διεργασία που προσπαθεί να κάνει συνάντηση μέσω τις exchange κάποια στιγμή θα επιστρέψει εξαιτίας (`while (getnanos() < TimeBound)` όπου θα γίνει timeout και θα επιστρέψει στην συνάρτηση που την κάλεσε και θα εκτελέσει μια push η μια pop. Δεν υπάρχει αλλού κάποιο spinning ή αναδρομική κλήση άρα θα έχουμε συνεχώς progress.

B. fine – grained synch

1. Ψευδοκώδικας.

```
1) #define EMPTY 1
2) #define WAITING 2
3) #define BUSY 3
4) #define CAPACITY 100
5)
6) //Each element of exchanger[] is initially <null, EMPTY>.
7) <T, state, exchange> exchanger[CAPACITY];
8) state = {EMPTY, WAITING, BUSY}
9) //Locks implemented with mutex lock
10)
11) shared Lock stack;
12)
13) T visit(T value, int range, long duration) {
14)     int el = randomnumber(range);
15)     return (exchange(exchanger[el], value, duration));
16) }
17)
18)
19)T exchange(shared <T, int> slot, T myitem, long timeout) {
20)    long timeBound = getnanos() + timeout;
21)    while (TRUE) {
22)      // if it is time for timeout, leave the exchanger
23)      if (getnanos() > timeBound) return TIMEOUT;
24)      lock(slot.exchange);
25)      <youritem, state, l> = slot;
26)      switch (state) {
27)          case EMPTY: // try to place your item in the slot and set state to WAITING
28)              slot = <myitem, WAITING, l>;
29)              unlock(l.exchange);
30)              while (getnanos() < TimeBound) { // spin until it is time for timeout
31)                  <youritem, state, l> = slot; // read slot
32)                  if (state == BUSY) {
33)                      slot = <null, EMPTY, l>; // if the exchange is complete
34)                      return youritem;
35)                  } // return the other process's item
36)              }
37)              // if no other thread shows up
38)              if (slot == myitem) { //linearization *
39)                  slot = <null, EMPTY, l>
40)                  return TIMEOUT
41)              } else {
42)                  lock(l.exchange);
43)                  <youritem, state, l> = slot;
44)                  slot = <null, EMPTY, l>;
45)                  unlock(l.exchange);
46)                  return youritem;
47)              }
48)          case WAITING: // some thread is waiting and slot contains its item
49)              slot = <myitem, BUSY, l>; //linearization *
50)              unlock(l.exchange);
51)              return youritem; // and return the item of the other process
52)          case BUSY: // two other threads are currently using the slot
53)              break; // the process must retry
54)      } // switch
55) }
56)}
```

Οι συναρτήσεις pop,push παραμένουν ίδιες με πρίν.

2. Ο αλγόριθμος είναι ο ίδιος με τον coarse – grained. Η διαφορά εδώ είναι:
   7) <T, state, exchange> exchanger[CAPACITY];
   Δηλαδή για κάθε slot έχουμε προσθέσει εσωτερικά ακόμα ένα attribute το οποίο θα είναι το lock του. Άρα όταν θέλουμε να κάνουμε lock το slot πχ 10 θα προσπαθούμε να κάνουμε lock την μεταβλητή του slot αυτού και όχι

ένα γενικό lock variable που θα είναι κοινό για όλες τις διεργασίες. Επομένως μπορούν παράλληλα να βρίσκονται παραπάνω διεργασίες στην κρίσιμη περιοχή αρκεί να spinnaroun σε διαφορετικο slot.

3. Ισχύουν τα ίδια linearization points με το α3.
4. Όμοια με το α4.
5. Όμοια με το α4.