VALGRIND

Έκανα τις εξής διορθώσεις στον κώδικα:

1. **Διόρθωση stack_init()**: Προστέθηκε malloc για την κατανομή μνήμης στη Stack, αποφεύγοντας τη μη αρχικοποιημένη χρήση της.
2. **Διόρθωση stack_pop()**:
    - Προστέθηκε σωστή απελευθέρωση μνήμης για τον τελευταίο κόμβο.
    - Διορθώθηκε το σενάριο όταν η στοίβα έχει μόνο έναν κόμβο.
3. **Διόρθωση stack_peek()**: Διορθώθηκε το invalid read, αφαιρώντας την πρόσβαση σε cur->next->data και επιστρέφοντας το cur->data.
4. **Διόρθωση stack_push()**: Διορθώθηκε η λανθασμένη συνθήκη while (cur != NULL), που προκαλούσε invalid read/write.
5. **Διόρθωση stack_merge()**: Τώρα απελευθερώνει σωστά τη μνήμη των αρχικών στοιβών.
6. **Διόρθωση stack_destroy()**: Απελευθερώνει σωστά όλους τους κόμβους και μετά τη στοίβα.

Δοκίμασε να τρέξεις ξανά το valgrind ./test και valgrind ./mem_errors για να επιβεβαιώσεις ότι τα memory leaks και τα errors έχουν εξαφανιστεί!

Διαδικασία Debugging με Valgrind για Ανίχνευση και Διόρθωση Σφαλμάτων Μνήμης

Η άσκηση ζητά να διορθώσεις τα memory errors και leaks στο αρχείο **faulty_stack.c** χρησιμοποιώντας το **Valgrind**. Ακολουθεί η αναλυτική διαδικασία για τον εντοπισμό και τη διόρθωση των σφαλμάτων:

1️⃣ Compilation (Μεταγλώττιση) του Κώδικα

Πριν τρέξεις το **Valgrind**, πρέπει να μεταγλωττίσεις το πρόγραμμα για να σιγουρευτείς ότι δεν υπάρχουν syntax errors:

make

Αν η μεταγλώττιση είναι επιτυχής, μπορείς να προχωρήσεις με τη δοκιμή.

2️⃣ Ανίχνευση Σφαλμάτων με Valgrind

Χρησιμοποιούμε το **Valgrind** για να ελέγξουμε το πρόγραμμα για memory leaks και σφάλματα προσβασιμότητας στη μνήμη.

Έλεγχος του mem_errors

Τρέχουμε το πρόγραμμα με **Valgrind**:

valgrind --leak-check=full --show-leak-kinds=all ./mem_errors

📌 **Τι κάνει αυτό;**

- --leak-check=full: Εμφανίζει λεπτομερείς πληροφορίες για τα memory leaks.
- --show-leak-kinds=all: Αναφέρει διαφορετικούς τύπους memory leaks (definitely lost, indirectly lost, etc.).

Αναλύοντας το Output του Valgrind:

- **Invalid Read/Write**: Το πρόγραμμα προσπαθεί να διαβάσει ή να γράψει σε απαγορευμένη μνήμη.
- **Use After Free**: Προσπάθεια πρόσβασης σε μνήμη που έχει ήδη αποδεσμευτεί.
- **Memory Leaks**: Μνήμη που δεσμεύτηκε αλλά δεν αποδεσμεύτηκε πριν από το τέλος του προγράμματος.

Έλεγχος των Unit Tests (test_faulty_stack.c)

Εκτελούμε τα **unit tests** με το Valgrind για να ελέγξουμε αν η στοίβα λειτουργεί σωστά:

valgrind --leak-check=full ./test

Αν δεις **Memory Leaks** ή **Invalid Reads/Writes**, τότε πρέπει να εστιάσεις στις συγκεκριμένες συναρτήσεις του faulty_stack.c.

3️⃣ Διόρθωση των Σφαλμάτων

📌 Διορθώσεις που ζητά η άσκηση:

1. stack_init() – Initialization Error

Σφάλμα: Η συνάρτηση δεν κάνει malloc για τη στοίβα, με αποτέλεσμα χρήση μη αρχικοποιημένης μνήμης.

✅ **Διόρθωση:**

```
Stack *stack_init(){
    Stack *s = malloc(sizeof(Stack)); // Δέσμευση μνήμης
    if (!s) return NULL; // Έλεγχος αποτυχίας malloc
    s->list = NULL;
    s->count = 0;
    return s;
}
```

---

2. stack_pop() – Memory Leak

Σφάλμα: Δεν αποδεσμεύει σωστά τον τελευταίο κόμβο.

✅ **Διόρθωση:**

```
int stack_pop(Stack *s){
    if(stack_empty(s))
        return INT_MIN;

    Node *cur = s->list, *prev = NULL;
    while (cur->next != NULL) {
        prev = cur;
        cur = cur->next;
    }
```

```
    int result = cur->data;
    free(cur); // Τώρα γίνεται σωστά η αποδέσμευση

    if (prev) prev->next = NULL;
    else s->list = NULL; // Αν υπήρχε μόνο ένας κόμβος, ενημερώνουμε τη στοίβα

    s->count--;
    return result;
}
```

---

3. stack_peek() – Invalid Read

Σφάλμα: Προσπαθεί να διαβάσει cur->next->data, ακόμα και αν cur->next == NULL.

✅ **Διόρθωση:**

```
int stack_peek(Stack *s){
    if(stack_empty(s))
        return INT_MIN;

    Node *cur = s->list;
    while(cur->next != NULL){
        cur = cur->next;
    }

    return cur->data; // Δεν προσπαθούμε να προσπελάσουμε cur->next->data
}
```

---

4. stack_push() – Invalid Read/Write

Σφάλμα: Ο βρόχος while(cur != NULL) τρέχει μέχρι να γίνει NULL, αλλά μετά προσπαθούμε να γράψουμε στο cur->next.

✅ **Διόρθωση:**

```
Stack* stack_push(Stack *s, int data){
    Node *node = create_stack_node(data);


    if (!s->list) {
        s->list = node;
    } else {
        Node *cur = s->list;
        while(cur->next != NULL){ // Πρέπει να τερματίζει στο τελευταίο στοιχείο
            cur = cur->next;
        }
        cur->next = node; // Σωστή προσθήκη κόμβου
    }


    s->count++;
    return s;
}
```

---

5. stack_merge() – Memory Leaks

Σφάλμα: Η νέα στοίβα δημιουργείται, αλλά δεν αποδεσμεύουμε τις αρχικές.

✅ **Διόρθωση:**

```
Stack* stack_merge(Stack *s1, Stack *s2){
    Stack *newStack = stack_init();

    Node *cur = s1->list;
    while(cur != NULL){
        newStack = stack_push(newStack, cur->data);
        cur = cur->next;
    }

    cur = s2->list;
    while(cur != NULL){
        newStack = stack_push(newStack, cur->data);
        cur = cur->next;
    }

    stack_destroy(s1); // Τώρα αποδεσμεύουμε τις αρχικές στοίβες
    stack_destroy(s2);

    return newStack;
}
```

---

6. stack_destroy() – Memory Leaks

Σφάλμα: Δεν αποδεσμεύει όλους τους κόμβους.

✅ **Διόρθωση:**

```
void stack_destroy(Stack *s){
    Node *cur = s->list, *next;
    while(cur != NULL){
        next = cur->next;
        free(cur);
        cur = next;
    }
    free(s);
}
```

4️⃣ Επαλήθευση ότι όλα είναι σωστά

Μετά τις αλλαγές, ξανατρέχουμε:

```
valgrind --leak-check=full ./test
valgrind --leak-check=full ./mem_errors
```

Αν όλα έχουν διορθωθεί, το Valgrind **δεν θα εμφανίσει memory leaks ή errors**. 🎯

📌 **Συμπέρασμα**

Η διαδικασία debugging με το **Valgrind** είναι ουσιαστική για την εύρεση:

- **Memory Leaks** (χαμένη μνήμη που δεν αποδεσμεύεται).
- **Invalid Reads/Writes** (προσπάθεια προσπέλασης μνήμης εκτός ορίων).
- **Use After Free** (χρήση αποδεσμευμένης μνήμης).
- **Double Free Errors** (διπλή αποδέσμευση).

Με τη σωστή χρήση του Valgrind, εντόπισες και **διόρθωσες όλα τα προβλήματα μνήμης στη στοίβα**!