מטלה 3

שלב 1

לפי הנחיות המטלה, הנחנו שהקלטים תקינים ולכן לא ביצענו בדיקות חריגה (כמו פסיקים כפולים, תווים שאינם מספרים וכו').

למימוש האלגוריתם השתמשנו בשיטת **Monotone Chain** (וריאנט של Graham scan), בעלת סיבוכיות זמן של $O(n \log n)$.

התוצאה המתקבלת מהאלגוריתם היא רשימת נקודות על המעטפת הקמורה, ולאחר מכן בוצע חישוב שטח על ידי נוסחת שטח פוליגון (שיטת shoelace).

הרצה על דוגמת הקלט שניתנה במטלה (4 נקודות) הניבה שטח של 1.5, כפי שנדרש.

מצורף צילום התוצאה:

```
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneDrive/הדובעה וחלוש/projects/OperationSystem/os_3/part1$ ./bin/ConvexHull
4
0,0
0,1
1,1
2,0
1.5
```

בנוסף מצורפת הדגמה עבור קלט ופלט עם קודקוד שאינו תקין במהלך ההזנה:

הפלט:

```
3
0,0
a,b
1,0
0,1
```

הקלט:

```
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneDrive/הדובעה וחלוש/projects/OperationSystem/os_3/part1$ ./bin/ConvexHull < input.txt
ERROR: Invalid point values.
0.5
```

שלב 2

בשלב זה מימשנו את אלגוריתם ה־Convex Hull בשתי גרסאות נפרדות:
האחת מבוססת על `std::deque`, והשנייה על `std::list`.

הרצנו את שתי הגרסאות על קובץ קלט המכיל 200 נקודות.
שתי הגרסאות החזירו תוצאה זהה של 943215.

- זמן הריצה של הגרסה עם `deque` היה יעיל יותר מאשר של הגרסה עם ה `list`.

מכך ניתן להסיק כי `deque` מהירה יותר ויעילה יותר, כנראה בזכות גישה ישירה ומהירה יותר לזיכרון לעומת `list`, שהיא מבנה מקושר.

מצורף צילום התוצאה:

```
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneDrive/שולחן העבודה/projects/OperationSystem/os_3/part2$ ./bin/ConvexHullDeque < input.txt
Area: 943215
Time (deque): 0.240565 ms
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneDrive/שולחן העבודה/projects/OperationSystem/os_3/part2$ ./bin/ConvexHullList  < input.txt
Area: 943215
Time (list): 0.449345 ms
```

שלב 3

בשלב זה הרחבנו את יישום מעטפת השיפוע (Convex Hull) כך שיתמוך בהזנת פקודות דינמית דרך הקלט הסטנדרטי (`stdin`). המשתמש יכול להזין פקודות אחת אחרי השנייה לצורך בניית גרף חדש, הוספה או הסרה של נקודות, וחישוב השטח של מעטפת השיפוע של הנקודות הנוכחיות.

התנהגות מיוחדת בעת הזנת גרף:

במהלך הזנת נקודות לאחר פקודת `Newgraph`, התוכנית נכנסת למצב "המתנה לנקודות".
במצב זה, כל שורה חייבת להיות בפורמט תקני של נקודה (`x, y`).
כל שורה שאינה תקינה — בין אם מדובר בפורמט שגוי, ערכים לא מספריים, או פקודה אחרת לגיטימית (כגון `CH`) — תגרום להדפסת הודעת שגיאה, אך **לא תפסיק את תהליך ההזנה**.

רק לאחר שהוזנו כל `N` הנקודות התקינות, התוכנית תצא ממצב ההזנה ותשוב לעיבוד פקודות רגיל.

דוגמה לפלט וקלט תקין:

```
Newgraph 3
0,0
1,0
0,1
CH
Newpoint 1,1
CH
Removepoint 0,0
CH
Newgraph 2
0,0
1,1
Newpoint 2,2
CH
Newgraph 2
1,2
2,3
CH
Newgraph a
CH
Newgraph 2
x,y
0,0
CH
```

```
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneD
0.5
1
0.5
0
0
ERROR: Invalid number in Newgraph.
0
ERROR: Invalid point values.
ERROR: Invalid point format.
```

הסבר:

בדוגמה שמעלה ניתן לראות את הפלט שהתקבל עבור קובץ קלט הכולל רצף של פקודות תקינות ושגויות.

- שורות 1-5: נבנה גרף עם שלוש נקודות, ופקודת CH מחזירה שטח 0.5.
- שורות 6-7: מוסיפות נקודה נוספת ומרחיבות את מעטפת השיפוע, פקודת CH אשר מראה תוצאה 1.
- שורות 8-9: מסירות נקודה ומחזירות את השטח ל־0.5 ופקודת CH אשר מדגימה את התוצאה.
- שורות 10-14: מוגדר גרף חדש עם שתי נקודות, אליו מתווספת נקודה שלישית, אך כולן על קו ישר, ולכן ופקודת CH מחזירה שטח 0.
- שורות 15-18: נבנה גרף נוסף עם שני נקודות (קו ישר) ופקודת CH , שוב שטח 0.
- שורה 19: ניסיון לבצע Newgraph עם פרמטר לא חוקי (a) – הודעת שגיאה מופיעה, והגרף הקודם נשאר שמור בזיכרון.
- שורה 20: פקודת CH לאחר מחשבת את המעטפת של הגרף הקודם, התוצאה עדיין 0.
- שורות 21-24: ניתנת פקודת Newgraph עם גודל 2, מוזנת נקודה לא תקינה (x, y) אשר מדפיסה שגיאה, לאחר מכן מוזנת נקודה תקינה, הפקודה CH נקראת בתור הזנת נקודה משום שלא הושלמה הזנה תקינה של כל הנקודות ולכן גם מדפיסה שגיאה.

השגיאות מוצגות באופן ברור, מבלי לשבור את הרצף הלוגי של התוכנית, הדוגמה מראה את שמירת המצב הרציפה והעמידות של המימוש בפני שגיאות קלט.

שלב 4

תמיכה בריבוי לקוחות (Shared Graph with Multi-Client Support)

מטרת השלב

בשלב זה שודרג השרת כך שיוכל לשרת מספר לקוחות במקביל, תוך שיתוף מצב גלובלי של גרף הנקודות, ובכך לאפשר לכל לקוח להוסיף נקודות, להסיר אותן, או לחשב את ה־Convex Hull, כאשר כל שינוי גרפי מתבצע על **גרף אחיד ומשותף** לכל החיבורים.

לצד זאת, נדרש גם להבטיח **התנהגות עקבית ובטוחה** כאשר לקוח מבצע פעולה "חלקית" או "מורכבת", כמו Newgraph, אשר דורשת הזנת מספר נקודות לפני שהשרת חוזר למצב רגיל.

פונקציונליות קיימת משלב קודם

השרת בשלב 3 כבר הכיל את המרכיבים הבאים:

- שימוש ב־select() לניהול מספר לקוחות בו־זמנית באמצעות multiplexing.
- מבני נתונים בסיסיים כמו point_set (רשימת הנקודות הקיימות), temp_points, והדגל waiting_for_graph.
- טיפול בפקודות CH, Removepoint, Newpoint, Newgraph דרך הפונקציה process_line.
- קלט טקסטואלי פשוט באמצעות recv ו־send.

לכן, התמיכה הראשונית בריבוי לקוחות כבר הוטמעה, אך הייתה חסרה לוגיקה שמנהלת נכון **התנהגות שיתופית** ו־**בקרת גישה לפקודות**.

שינויים שבוצעו בשלב 4

1. זיהוי לקוח יוזם (newgraph_owner_fd)

הוסף משתנה חדש מסוג int, אשר מזהה את ה־file descriptor של הלקוח ששלח את פקודת Newgraph. משתנה זה מאפשר לוודא שרק הלקוח שיזם את הגרף יכול להמשיך ולשלוח את הנקודות. לקוחות אחרים שינסו לשלוח פקודות יידחו אוטומטית עם הודעת שגיאה "BUSY".

2. טיפול מדויק בפר לקוח (<unordered_map<int, ClientState>)

הוגדר מבנה ClientState לכל לקוח, הכולל buffer פנימי (inbuf) לאגירת שורות קלט עד קבלת תו \n. כך ניתן לנהל תקשורת אמינה עם לקוחות גם כאשר מגיעות מספר שורות יחד (או חלקיות).

3. ניהול מצב Newgraph משופר

השרת שודרג כך שכאשר מתקבלת פקודת Newgraph, הוא נכנס למצב חסום (waiting_for_graph) בו מתקבלות רק נקודות מהלקוח הרלוונטי. כל שאר הפעולות נדחות, כולל מלקוחות אחרים. לאחר קבלת כל הנקודות (לפי המספר שהוגדר), הגרף נרשם כקיים ומוחזר "GRAPH_LOADED" ללקוח.

4. איפוס אוטומטי במקרה של ניתוק

במקרה בו לקוח מתנתק במהלך שלב Newgraph מבלי שהשלים את הכנסת הנקודות – השרת מזהה זאת ומבצע איפוס של המצב (waiting_for_graph = false, ניקוי temp_points, איפוס newgraph_owner_fd), ובכך מונע תקיעות לוגית.

5. פידבקים חדשים ללקוח

- "GRAPH_LOADED" – מוחזר לאחר קליטת כל הנקודות הנדרשות ב־Newgraph.
- "BUSY" – מוחזר ללקוחות שמנסים לשלוח פקודות בזמן שלקוח אחר עדיין נמצא בתהליך Newgraph.

דוגמת הרצה:

- לקוח א' הגדיר גרף חדש עם 4 נקודות, ולאחר מכן קיבל שטח 1.5 מה־CH.
- לקוח ב' התחבר במקביל, שלח CH וקיבל גם הוא את אותו ערך – מה שמעיד על גרף משותף.
- לקוח ב' הוסיף נקודה (Newpoint 1,2) וחישוב CH מחודש החזיר 2.0.

```
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneDrive/שלוש וחצי הדובעה/projects/OperationSystem/os_3/part4$ make
mkdir -p bin
g++ -std=c++17 -Wall -Wextra -O2 -Iinclude -o bin/ConvexHullServer src/GeometryUtils.cpp main/Main.cpp
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneDrive/שלוש וחצי הדובעה/projects/OperationSystem/os_3/part4$ ./bin/ConvexHullServer
```

```
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneDrive/שלוש וחצי הדובעה/projects/OperationSystem$ nc localhost 9034
Newgraph 4
OK
0,0
OK
1,1
OK
1,0
OK
0,2
GRAPH_LOADED
CH
1.5
```

```
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneDrive/שלוש וחצי הדובעה/projects/OperationSystem$ nc localhost 9034
CH
1.5
Newpoint 1,2
OK
CH
2
```

שלב 5

בשלב זה מימשנו תבנית Reactor כמנגנון כללי לניהול אירועים אסינכרוניים מבוסס fd.

רכיבי מנגנון הריאקטור שלנו:

1. מפת handlers

```
;std::map<int, reactorFunc> fd_to_func
```

מפה זו קושרת בין כל file descriptor (לדוגמה: 4, 5, 6...) לבין פונקציית טיפול ייעודית שמוגדרת כ־reactorFunc.

למשל, אם לקוח מתחבר דרך fd=4, הפונקציה שנרשמה עבור fd=4 תיקרא אוטומטית כשיגיע מידע.

2. מנעול – mutex

```
;std::mutex reactor_mutex
```

המנעול משמש להגנה על **המפה עצמה** בזמן עדכון. אם שני threads היו מוסיפים בו-זמנית fd חדש – תיווצר קריסה או גישה לא עקבית. לכן כל פעולה של הוספה/הסרה/בדיקה מתבצעת תחת `std::lock_guard<std::mutex>`.

3. ה־thread של הריאקטור

זהו thread שרץ בלולאה אינסופית, ומבצע:

- `select()` על כל ה־fds הפעילים.
- ברגע ש־fd מוכן לקריאה – מזמן את הפונקציה המתאימה ממפת ה־handlers.

4. API פנימי

הוספנו פונקציות כמו:

```
void addFdToReactor(int fd, reactorFunc func);
void removeFdFromReactor(int fd);
void startReactor();
void stopReactor();
```

כך שכל חלק בקוד יכול **לרשום או להסיר** fd בקלות – מבלי לדעת איך select עובד בפנים.

שלב 6

השינוי העיקרי שביצענו הוא **החלפה של טיפול ידני בלקוחות** (כגון `accept`, `recv`, `send` שנעשים בלולאה ראשית אחת) במבנה מודולרי שמבוסס על *ריאקטור*.

במקום שנאזין ישירות לכל fd בכל רגע, הריאקטור **מנהל עבורנו את הפיקוח על כל ה־file descriptors**, ומזמן את פונקציית הטיפול (callback) המתאימה כשמגיע מידע חדש.

איך מתבצע טיפול בלקוח עכשיו?

כאשר לקוח חדש מתחבר:

1. הריאקטור מזהה את `fd=4` מוכן ל־read.
2. הוא מוצא את הפונקציה המתאימה במפה עבור fd=4.
3. קורא לה.
4. בפונקציה הזו מתבצע `recv` → פענוח פקודה → שליחת תגובה.

תוספות הדפסות לשרת לנוחות ודיבאג:

לצורך ניטור ברור של פעולות השרת במהלך הרצה ובדיקות, הוספו הדפסות Debug שמסייעות להבין את התקשורת בזמן אמת. ההדפסות כוללות:

- `<New client accepted: <fd` – כאשר לקוח חדש מתקבל.
- `<Received from fd <fd>: <command` – כאשר מתקבלת פקודה מהלקוח.
- `<Processing line: <command` – הדפסת הפקודה לפני עיבודה.
- `<Response: <response →` – הדפסת התגובה שנשלחת ללקוח לאחר עיבוד.

דוגמת הרצה:

- לקוח א' שלח CH וקיבל 0 כתשובה, לאחר מכן הגדיר גרף חדש עם 3 נקודות.
- לקוח ב' התחבר במקביל, שלח CH וקיבל BUSY כתשובה - כלומר השרת באמצע פעולה (באמצע לקבל את הגרף החדש) לאחר מכן שלח שוב CH וקיבל את הערך המתאים לגרף 2 – מה שמעיד על גרף משותף.
- לקוח ב' הוסיף נקודה (Newpoint 2,2) לגרף.
- לקוח ב' התנתק ואז גם לקוח א'.

לקוח א:

```
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneDrive/שלישי ורביעי/projects/OperationSystem$ nc localhost 9034
CH
0
Newgraph 3
OK
0,0
OK
0,2
OK
2,0
GRAPH_LOADED
^C
```

לקוח ב:

```
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneDrive/שלישי ורביעי/projects/OperationSystem$ nc localhost 9034
CH
BUSY
CH
2
Newpoint 2,2
OK
^C
```

שרת:

```
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneDrive/הדובעה ,חקלוש/projects/OperationSystem/os_3/part6$ ./bin/ConvexHullServer
Server is running. Press Ctrl+C to exit.

New client accepted: 4

Received from fd 4: CH

Processing line: CH
 → Response: 0

Received from fd 4: Newgraph 3

Processing line: Newgraph 3
 → Response: OK

New client accepted: 5

Received from fd 5: CH

Processing line: CH
 → Response: BUSY

Received from fd 4: 0,0

Processing line: 0,0
 → Response: OK

Received from fd 4: 0,2

Processing line: 0,2
 → Response: OK

Received from fd 4: 2,0

Processing line: 2,0
 → Response: GRAPH_LOADED
```

```
Received from fd 5: CH

Processing line: CH
 → Response: 2

Received from fd 5: Newpoint 2,2

Processing line: Newpoint 2,2
 → Response: OK

Client 5 disconnected or error occurred (recv=0). Closing fd.
Client 4 disconnected or error occurred (recv=0). Closing fd.
```

שלב 7

שלב 7 – תמיכה במספר לקוחות במקביל באמצעות תהליכונים (Threads) ונעילות (mutex)

בשלב זה עברנו ממערכת ריאקטיבית (reactor) לתכנון מבוסס תהליכונים (threads), שבו כל לקוח מקבל טיפול עצמאי אך עדיין כולם פועלים על אותו גרף משותף.

כדי למנוע בעיות עקב ריבוי גישות לגרף, השתמשנו בנעילה (mutex) שמבטיחה שהשינויים בו יתבצעו בצורה בטוחה ובתיאום.
כך הצלחנו לשמור על ההתנהגות התקינה של השרת גם תחת עומס של לקוחות בו־זמניים.

✅ שינויים עיקריים:

1. **מודל thread לכל לקוח**

- עם קבלת חיבור חדש (accept), מופעל std::thread שמטפל בלולאה עבור אותו לקוח בלבד.
- התהליכון מפעיל את פונקציית הטיפול בלקוח, ומיד מנותק (detach) כך שלא יכביד על הלולאה הראשית.

2. **הגנה על מבנה הנתונים הגלובלי באמצעות std::mutex**

- מאחר שכל הלקוחות עובדים עם אותו גרף משותף (המשתנה point_set), נדרש להגן על הגישה אליו כדי למנוע קריאות/כתיבות במקביל.

הוספנו mutex בשם graph_mutex, וכל גישה לגרף עוטפה ב־:std::lock_guard<std::mutex>

cpp
CopyEdit
```
(std::lock_guard<std::mutex> lock(graph_mutex;
```

-

3. **שמירה על מבנה הקלט של כל לקוח**

- נשמרה המפה clients עם מבנה ClientState, כך שניתן לצבור שורות קלט עד לקבלת שורת פקודה שלמה (מסתיימת ב־\n).

4. **טיפול בניתוקים**

- במקרה של ניתוק או שגיאה מצד הלקוח (recv = 0), נסגר החיבור ונתוני הלקוח נמחקים.
- אם לקוח נותק באמצע הגדרת גרף חדש (Newgraph), ההגדרה מבוטלת והמצב הגלובלי מתאפס.

דוגמת הרצה:

- לקוח א' שלח CH וקיבל 0 כתשובה, לאחר מכן הגדיר גרף חדש עם 3 נקודות שלח CH וקיבל 2.
- לקוח ב' התחבר במקביל, שלח CH לפני תחילת שליחת הגרף של א' וקיבל 0 כתשובה, לאחר מכן שלח שוב תוך כדי שליחת הגרף של א' וקיבל BUSY כתשובה - כלומר השרת באמצע פעולה (באמצע לקבל את הגרף החדש) ניסה להוסיף נקודה בעזרת Newpoint 2,2 ושוב נתקל בBUSY כתשובה וזאת בגלל שלקוח א' עדיין לא סיים להזין את הגרף.
- לקוח ב' הוסיף נקודה (Newpoint 2,2) לגרף ובעזרת CH קיבל 4.
- לקוח א' בעזרת CH קיבל 4 (כלומר הגרפים מתממשקים טוב).
- לקוח ב' הסיר נקודה מהגרף (Removepoint 0,0).
- לקוח א' בעזרת CH קיבל 2 (כלומר הגרפים שוב מתממשקים טוב).
- שני הלקוחות התנתקו.

```
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneDrive/הדובעה וחלוש/projects/OperationSystem$ nc localhost 9034
CH
0
Newgraph 3
OK
0,0
OK
0,2
OK
2,0
GRAPH_LOADED
CH
2
CH
4
CH
2
^C
```

```
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneDrive/הדובעה וחלוש/projects/OperationSystem$ nc localhost 9034
CH
0
CH
BUSY
Newpoint 2,2
BUSY
Newpoint 2,2
OK
CH
4
Removepoint 0,0
OK
^C
```

שלב 8

מימוש ספריית Proactor

בשלב זה יצרנו ספרייה נפרדת למימוש תבנית **Proactor**, המפעילה **thread ייעודי עבור כל לקוח חדש**. הספרייה מספקת ממשק פשוט: הפעלת פרואקטור (`startProactor`) ועצירתו (`stopProactor`) תוך שימוש ב־`pthread_t`.

המימוש גנרי ואינו תלוי בלוגיקת השרת, ומאפשר להריץ פונקציית טיפול לכל לקוח בנפרד. שלב זה שומר על עבודה עם **גרף משותף אחד**, תוך שימוש חיצוני ב־`mutex` להגנה על הנתונים. זהו שלב מקדים לקראת שילוב מלא עם Reactor בשלב הבא.

שלב 9

שילוב תבניות Reactor ו־Proactor

בשלב זה שילבנו לראשונה את שתי תבניות התכנות – **Reactor** ו־**Proactor** – לשרת אחד, שמסוגל להאזין לחיבורים חדשים בגישת Reactor ולטפל בכל לקוח חדש באמצעות Thread עצמאי בגישת Proactor.

מטרת השלב:

להפעיל שרת שבו:

- ה־**Reactor** מאזין לחיבורים נכנסים (accept) ומזהה מתי מגיע לקוח חדש.
- לכל לקוח חדש, ה־**Proactor** מפעיל Thread חדש (באמצעות pthread_t) שמטפל בתקשורת מול אותו לקוח – כלומר, קריאה, עיבוד ושליחה של תגובות.

עקרונות מימוש:

- **Reactor** רץ ברקע באמצעות thread ייעודי, ומגיב לאירועים מה־socket של ה־listener (כמו חיבור חדש).
- **Proactor** מופעל עבור כל לקוח חדש – והוא יוצר Thread-Per-Client שמבצע את עבודת התקשורת בפועל.
- **Mutex** מגן על מבני הנתונים המשותפים (כמו point_set, temp_points, waiting_for_graph) כדי למנוע תנאי תחרות (Race Conditions).
- כל השינויים ברשימת הנקודות, חישוב המעטפת הקמורה (Convex Hull) ותפעול פקודות כמו CH, Newpoint, Newgraph נעשים מתוך threads שמנוהלים בפרואקטור, תוך הקפדה על סנכרון בעזרת std::mutex.

יתרונות המימוש המשולב:

- ניהול יעיל של **חיבורים מרובים** – בזכות תבנית ה־Reactor, שנועדה לטפל באירועים על קבצי socket באופן לא חוסם.
- ביצועים טובים בעיבוד **מקבילי של לקוחות** – ה־Proactor יוצר thread חדש לכל לקוח, מה שמאפשר טיפול עצמאי ונוח בכל לקוח, ללא תלות באירועים נוספים.
- קוד מודולרי ומופרד: תבנית ה־Reactor עוסקת באיתור חיבורים חדשים בלבד, ותבנית ה־Proactor שולטת ב־lifecycle של הלקוח מהרגע שחובר.

אתגרים בולטים:

- הצורך **לסנכרן גישה ל־graph המשותף** בין threads שונים, דבר שנפתר ע"י שימוש עקבי ב־std::lock_guard.
- מימוש נכון של הממשק בין **Reactor ל־Proactor**: בעת קבלת לקוח חדש, נדרש להעביר את ה־socket לפונקציית thread מתאימה ולשמור על ממשק פשוט וברור בין החלקים.

דוגמת הרצה:

- לקוח א' שלח CH וקיבל 0 כתשובה, לאחר מכן הגדיר גרף חדש עם 3 נקודות שלח CH וקיבל 0.5.
- לקוח ב' התחבר במקביל, שלח CH לאחר שליחת הגרף של א' וקיבל וקיבל 0.5 כתשובה.
- לקוח ב' הוסיף נקודה (Newpoint 1,1) לגרף ובעזרת CH קיבל 1.
- לקוח א' בעזרת CH קיבל 1 (כלומר הגרפים מתממשקים טוב).
- לקוח א' החל להגדיר גרף חדש.
- לקוח ב' בעזרת CH קיבל BUSY (כלומר זיהוי טעינת הגרף של א' ואי מתן מענה).

- לקוח א' התנתק באמצע הגדרת הגרף.
- לקוח ב' בעזרת CH קיבל 1 כלומר הגרף הישן נתעט מחדש לאחר שלקוח א התנתק.
- לקוח ב' התנתק.

לקוח א':

```
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneDrive/שלוש ורבעה/projects/OperationSystem$ nc localhost 9034
CH
0
Newgraph 3
OK
0,0
OK
0,1
OK
1,0
GRAPH_LOADED
CH
0.5
CH
1
Newgraph 4
OK
1,1
OK
^C
```

לקוח ב':

```
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneDrive/שלוש ורבעה/projects/OperationSystem$ nc localhost 9034
CH
0.5
Newpoint 1,1
OK
CH
1
CH
BUSY
CH
1
^C
```

שרת:

```
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneDrive/הדובנה וחילוש/projects/OperationSystem/os_3/part9$ ./bin/ConvexHullServer
Server is running. Press Ctrl+C to exit.

New client accepted: 4

New client accepted: 5

Received from fd 4: CH

Processing line: CH
 → Response: 0

Received from fd 4: Newgraph 3

Processing line: Newgraph 3
 → Response: OK

Received from fd 4: 0,0

Processing line: 0,0
 → Response: OK

Received from fd 4: 0,1

Processing line: 0,1
 → Response: OK

Received from fd 4: 1,0

Processing line: 1,0
 → Response: GRAPH_LOADED

Received from fd 5: CH

Processing line: CH
 → Response: 0.5

Received from fd 4: CH

Processing line: CH
 → Response: 0.5
```

```
Received from fd 5: Newpoint 1,1

Processing line: Newpoint 1,1
 → Response: OK

Received from fd 4: CH

Processing line: CH
 → Response: 1

Received from fd 5: CH

Processing line: CH
 → Response: 1

Received from fd 4: Newgraph 4

Processing line: Newgraph 4
 → Response: OK

Received from fd 4: 1,1

Processing line: 1,1
 → Response: OK

Received from fd 5: CH

Processing line: CH
 → Response: BUSY

Client 4 disconnected or error occurred (recv=0). Closing fd.
Graph construction aborted (owner disconnected).
Received from fd 5: CH

Processing line: CH
 → Response: 1

Client 5 disconnected or error occurred (recv=0). Closing fd.
^C
```

שלב 10

ניטור שטח ה-Convex Hull באמצעות Thread נפרד

בשלב זה הוספנו יכולת ניטור בזמן אמת לשטח מעטפת ה־Convex Hull באמצעות **thread ייעודי**. המטרה הייתה לזהות מקרים שבהם שטח המעטפת **עולה או יורד מ-100 יחידות**, ולבצע הדפסה מתאימה לשרת.

מטרת השלב:

- לעקוב אחר שינויי שטח ה־CH (רק בבקשת חישוב מצד לקוח).
- להדפיס לוג ברגע שהשטח חוצה את סף ה־100 יחידות (מלמעלה או מלמטה).
- לשלב זאת כחלק מהתנהגות הרגילה של השרת מבלי לפגוע בריבוי הלקוחות ובטיפול הבקשות.

מימוש:

- הוגדרו משתני סנכרון:
  - pthread_mutex_t cond_mutex ו־pthread_cond_t cond לנעילה והמתנה.
  - bool at_least_100 לבדיקת תנאי ההדפסה.
  - std::atomic<bool> monitor_running לבקרת עצירת השרשור.
- נפתח **thread חדש** בשם monitor_thread שמאזין לשינויים:
  - הוא ממתין ל־pthread_cond_wait.

אם השטח יחצה את סף ה־100 יחידות — תודפס ההודעה הרולנטית:

- מנגנון ההדפסה מופעל מתוך פקודת "CH".

At Least 100 units no longer belongs to CH (חציה למטה).

At Least 100 units belongs to CH (חציה למעלה).

בדיקות:

- השרת הופעל ולקוח שלח פקודות להוספת נקודות ויצירת CH.
- בעת חישוב CH שהחזיר ערך **100 או יותר**, ההודעה הרצויה הופיעה בלוג השרת.
- גם קריאות חוזרות ל־"CH" לא גרמו להדפסות מיותרות, כל עוד הערך נשאר מעל 100.
- בעת שינוי הנקודות כך שהשטח ירד מתחת ל־100 והוחזר ל־100 – התקבלה הדפסה חוזרת, כנדרש.

דוגמת הרצה:

- לקוח א' הגדיר גרף חדש עם 3 נקודות שלח CH וקיבל 50.
- לקוח א' הוסיף נקודה (Newpoint 10,10) לגרף ובעזרת CH קיבל 100.
- לאחר החזרת ה CH השרת הדפיס At Least 100 units belongs to CH (כנדרש).
- לקוח א' ביקש שוב CH וקיבל 100.
- השרת לא הדפיס שוב את השורה (לא בוצע מעבר של הגבול).
- לקוח א' הסיר נקודה (Removepoint 0,0) מגרף ובעזרת CH קיבל 50.
- לאחר החזרת ה CH השרת הדפיס At Least 100 units no longer belongs to CH (כנדרש).
- לקוח א' ביקש שוב CH וקיבל 50.
- השרת לא הדפיס שוב את השורה (לא בוצע מעבר של הגבול).
- לקוח א' התנתק.

```
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneDrive/שולחן העבודה/projects/OperationSystem$ nc localhost 9034
Newgraph 3
OK
0,0
OK
10,0
OK
0,10
GRAPH_LOADED
CH
50
Newpoint 10,10
OK
CH
100
CH
100
Removepoint 0,0
OK
CH
50
CH
50
^C
```

```
orbibi@DESKTOP-6ENJ06D:/mnt/c/Users/LENOVO/OneDrive/שולחן העבודה/projects/OperationSystem/os_3/part10$ ./bin/ConvexHullServer
Server running on port 9034. Press Ctrl+C to exit.

New client accepted: 4
Received from fd 4: Newgraph 3

Processing line: Newgraph 3
 → Response: OK

Received from fd 4: 0,0

Processing line: 0,0
 → Response: OK

Received from fd 4: 10,0

Processing line: 10,0
 → Response: OK

Received from fd 4: 0,10

Processing line: 0,10
 → Response: GRAPH_LOADED

Received from fd 4: CH

Processing line: CH
 → Response: 50

Received from fd 4: Newpoint 10,10

Processing line: Newpoint 10,10
 → Response: OK

Received from fd 4: CH

Processing line: CH
 → Response: 100

At Least 100 units belongs to CH
```

```
Received from fd 4: CH

Processing line: CH
 → Response: 100

Received from fd 4: Removepoint 0,0

Processing line: Removepoint 0,0
 → Response: OK

Received from fd 4: CH

Processing line: CH
 → Response: 50

At Least 100 units no longer belongs to CH
Received from fd 4: CH

Processing line: CH
 → Response: 50

Client 4 disconnected or error occurred (recv=0). Closing fd.
^C
```