

תרגיל רטווב #3



ניהול הקצאת זכרון דינמית

הקדמה

שיםו לב – מקורות הקוד תבדק. העתקת שיעורי בית היא עבירה משמעת בטכניון על כל המשתמש מכך.

בתרגיל זה נמשג גרסת משלנו לפונקציות המוכרכות malloc, free ואחרות מ-h.h. על מנת ללמידה בצורה מעמיקה יותר על מרחב הזיכרון של תחילה, מנגנוני הקצתה זכרון ומעקב אחריהם, יעילות ההקצתה, תקשורת עם מערכת הפעלה ועוד.

כמה טיפים:

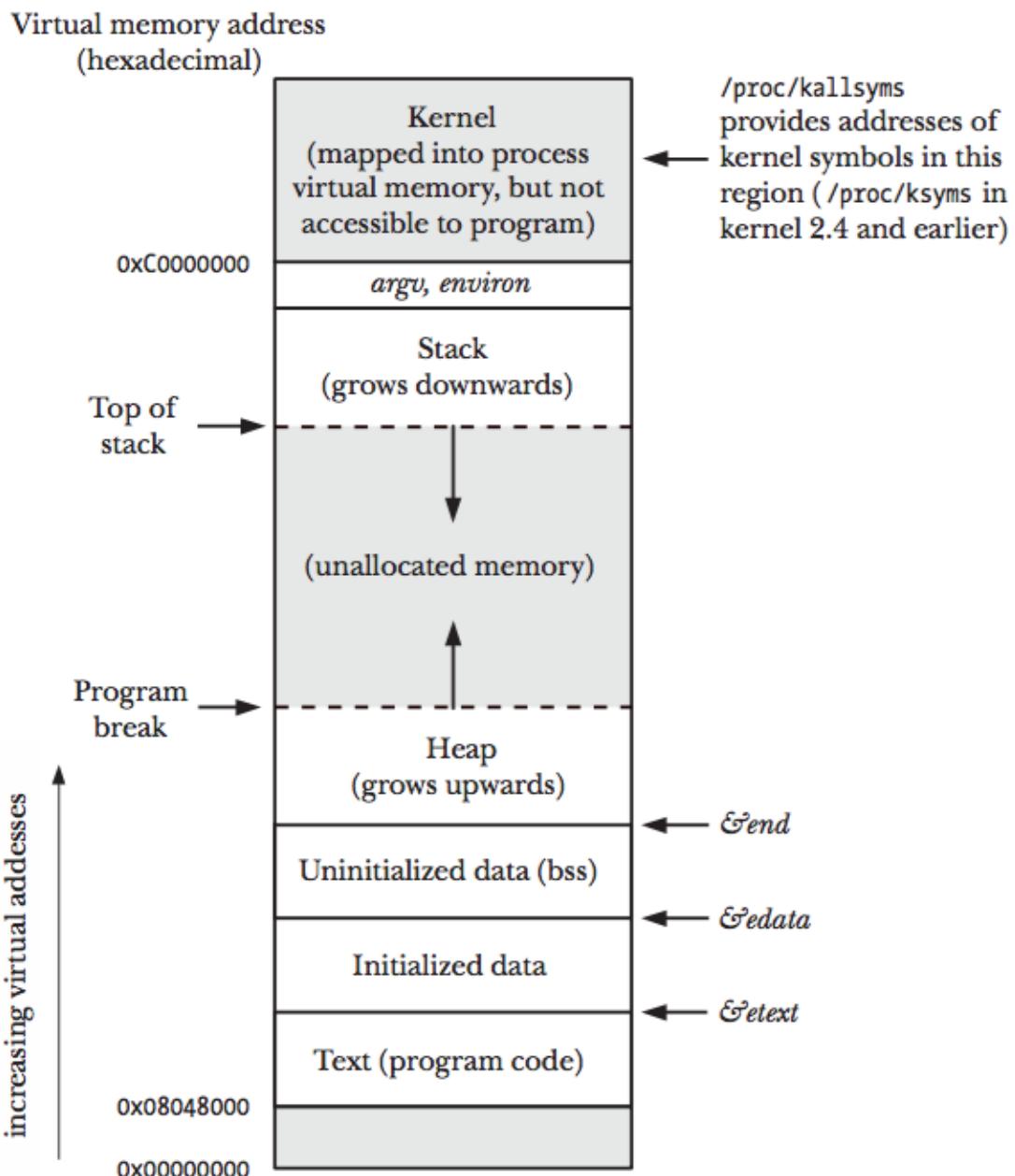
1. לפני ששאלים שאלות בפורים – עשו מאמץ לוודא שהתשובה לשאלתכם לא כתובה בהנחות התרגיל/שאלתם כבר בפורים/בקבוצה של הקורס. דגש מיוחד לתרגיל זה – שאלות כמו "האם זה בסדר שנעלו אTEX בזמן ש-Y..." הן חלק ממיומש הנדרש בתרגיל.
2. בחלק הבא של המספר תמצאו רקע תיאורטי לתרגיל ולאחריו מfcn התחלו לכתוב קוד.
3. **תכנו קודם!** שרטטו תרשימים זרימה כללי של התוכנית. מה הם מבני הנתונים הנדרשים? איך נממש אותם? הגדירו הצורות של classes/structs, תכנו פונקציות שתרצו למשר ורק אח"כ תחלו לכתב. ברגע שמתחייבים לימוש מסויים שלא לוקח בחשבון את אחת הדרישות, קsha הרבה יותר לשנות.
4. למדו להשתמש ב-gdb – בקורס ממ"ת יש סדנא עם כל מה שצריך לדעת על gdb, וזה כל'i מאד שימושי להבין מה מצב של כל משתנה בכל זמן, להשתמש ב-breakpoints וכו'.

בהצלחה!

רקע תיאורי

הערה – החומר זהה מופיע ביפורט בתרגול 8 (זיכרון וירטואלי #2).

מבנה הזיכרון הווירטואלי של התהיליך בلينוקס נראה סכמטית כך:



אנחנו נתענייןespecially ב-heap. הקצה העליון של heap נקרא **program break** של התהיליך והוא יכול לגדול ולקטן בצורה דינמית בהתאם לדרישות התהיליך. ה-stack גם יכול לגדול ולקטן בצורה דינמית, אבל בדרך כלל שלא מעורבות המשטמש לפני ריצת התוכנית ע"י הקומפайлר (קיפול ודחיפה של המחסנית בקריאה לפונקציות למשל). ה-heap, heap, מנהל ידנית ודינמית ע"י קראיות המערכת brk/sbrk שմבצע המשטמש במהלך ריצת התוכנית.

במהלך ריצה של התהיליך heap גדל כלפי מעלה בהקצת זיכרון ע"י פונקציות כמו malloc, וזה מתבטא בהגדלה של **program break**. עם זאת, לא כל קראיה ל-free יכולה להקטין בהכרח את ה-

– למשל, אם נשחרר זכרון שנמצא מתחת לזכרון אחר שעדין מוקצה, לא יוכל להקטין את ה- program break ותיזכר פרוגרמנטי. עם זאת, כן יוכל להקטין את הזכרון ששוחרר בפעם הבאה שהתהליך קורא ל-`malloc`.

שימוש לב - לתרגיל שלושה חלקים

קריאות מערכת רלוונטיות:

קריאות המערכת הבאות יישמשו אוטנו בכתיבת התרגיל:

```
int brk(void* addr);  
void* sbrk(size_t increment);
```

יש לקרוא את ה-`pages man` שלhn לעומך לקרוא מימוש התרגום – שימוש לב לפונקציונליות שלהם, ערכיו החזרה, קביעת ערכים עבור `errno` וכו'.

חישבו והיעזרו באינטרנט מה הפקודה `(0)brk` מוחזרה? מה ההשפעה שלה על ה-`PB`?

טיפ – צוות הקורס משתמש בפקודה זו לבדיקת נכונות הפתורן שלכם.

פרטים נוספים למימוש:

נרצה לייצר מנגנון לניהול ושחרור דינמי של זכרון ע"י הפונקציות `malloc` (והוריאציות שלה) ו-`free`. בתרגום намמש מנגנון פשוט יותר מה-`buddy system` להקצתה ושחרורם בлокים, שיפעל כך:

1. התוכנית תחזיק רשימה מקושרת של בлокים שהוקצו.
2. לפניה שבוצעו הקצאות, הרשימה תהיה ריקה.
3. כאשר מתבצעת הקצתה זכרון ע"י `malloc` או וריאנטים שלה:
 1. ראשית יש לנסות להחזיר בлок קיים וחופשי כלשהו שכבר הוקצה ונמצא מתחת ל-`program break`. רק במידה ולא קיים בлок מספיק גודל, יש להגדיל את מרחב הזכרון הדינמי של התהליך.
 2. אם מקצים בлок קיים, יש לבחור את הבלוק להקצתה בשיטת ה-`best fit` – כלומר, הבלוק שגודלו גדול/שווה לגודל הרצוי אבל הכל proximity קרוב לגודל הרצוי. למשל אם נדרש הקצותות 1KB יש בлокים פנויים (לא סמוכים) בגודל 2KB ו-4KB, יש לוודא שהבלוק של 2KB יבחר.
4. כאשר מתבצע שחרור זכרון ע"י `free`:
 1. במידה ולאחר השחרור קיימים שני בлокים חופשיים סמוכים, יש לאחד אותם לבלוק אחד גודל יותר.
 2. אם ניתן לשחרר זכרון למערכת הפעלה (כמובן, הבלוק בקצתה מרחב הזכרון), יש לבצע זאת כחלק מהפונקציה `free` ולשחרר את מקסימום הזכרון האפשרי.

חלק א'

יש למשתמש את הפונקציות הבאות. חלק מההנחיות מצלות ביחס לדרישות האמיתיות על הפונקציות, למשל אלו שתופיעו ב-pages.man.
הערה – "וריאנט של malloc" הוא .customMalloc/customCalloc/customRealloc

```
:void* customMalloc(size_t size)
```

1. מוקצה size בתים ומחייב מצביע לתחילת הזיכרון המוקצה.
2. יש לוודא כי הפונקציה לא מוקצת זיכרון כשיין צריך לכך (יש מספיק זיכרון רציף ופנוי מתחת ל- program break).
3. הפונקציה תבחר בлок פנוי בשיטת-best fit.
4. הפונקציה לא תבצע שום פעולה על הזיכרון שניית לה לפני שתחזיר אותו לשימוש.

```
:void customFree(void* ptr)
```

1. הפונקציה מקבל מצביע שהתקבל ע"י וריאנט של malloc ותשמן את הזיכרון שיופיע לפניו.
2. במידת האפשר, הפונקציה תאחד בלוקים.
3. במידת האפשר, לאחר השחרור הפונקציה תקטין את ה-program break של התהיליך כל שניית.
4. אם ptr הוא NULL, הפונקציה לא תבצע שום פעולה הקשורה לזכרון ותדפיס שגיאה:
`<free error>: passed null pointer`
5. על הפונקציה לוודא שהמצבי שהועבר אליה הוא מצביע שניית ע"י וריאנט של malloc. במידה והמצבי שהועבר הוא מצביע ל-stack, code או כל כתובות אחרת שאינה על ה-heap, על הפונקציה לא לבצע אף פעולה הקשורה לזכרון ולהדפיס שגיאה:

```
<free error>: passed non-heap pointer
```

```
:void* customCalloc(size_t nmemb, size_t size)
```

1. מוקצה nmemb אלמנטים שנודל כל אחד מהם בגודל size בתים, לסה"כ size * nmemb בתים, וכותבו את הערך 0 לכל אחד מהבטים הללו.
2. יש לוודא כי הפונקציה לא מוקצת זיכרון כשיין צריך לכך (יש מספיק זיכרון רציף ופנוי מתחת ל- program break).

המשך בעמוד הבא.

:void* customRealloc(void* ptr, size_t size)

1. הפונקציה מקבלת מצביע ptr וגודל size בbytes. נסמן ב-old את הגודל שהוקצה לptr במקור. מטרת הפונקציה היא לשנות את הגודל של הבלוק הנוכחי אליו ptr מצביע, בהתאם להגדרות הבאות.
2. על הפונקציה לוודא שהמצביע שהועבר אליה הוא מצביע שניית ע"י וריאנט של malloc. במידה והמצביע שהועבר הוא מצביע ל-stack, או כל כתובות אחרת שאינה על heap, על הפונקציה לא לבצע אף פעולה הקשורה לזכרון, להחזיר NULL ולהדפיס שגיאה:

<realloc error>: passed non-heap pointer

3. אם `old_size < size`, המטרה היא לשחרר זכרון (אם אפשר). במידת האפשר בהתאם לconiוגרציית הבלוקים והגדלים, הפונקציה תנסה לשחרר את הזכרון שבין `size` ל-`old_size`. אם הפונקציה לא מצילה לשחרר את הבלוקים בצורה המתאימה, יתבצעו הפעולות הבאות:

1. הפונקציה תקצת בלוק חדש בגודל `size` (הקטן מה-`old_size`)
 2. תעתק לבlok החדש את תכני הבלוק הישן שבין `size` לבין `old_size` בתים (כלומר, תשמיט מהבלוק החדש את החלק "שמייל" ל-`(size - old_size)`)
 3. תשחרר את הבלוק הישן
 4. תחזר מצביע לבlok החדש
4. אם `old_size > size`:
 1. הפונקציה תקצת בלוק חדש בגודל `size`
 2. תעתק לבlok החדש את כל תכני הבלוק הישן
 3. תשחרר את הבלוק הישן
 4. תחזר מצביע לבlok החדש
5. אם `NULL == ptr`, הפונקציה תהיה שולחה ל-`malloc(size)`.

במידה ו-`sbrk/brk` נכשלות, ושהזחות האם הקרה אליהם הייתה לא נכונה (למשל עם פרמטרים בעיתים) או שנגמר למערכת הזיכרון. במידה ונגמר למערכת הזיכרון, יש לשחרר את כל הזיכרון, להדפיס את השגיאה הבאה ולאחר מכן לצאת ע"י `:exit(1)`:

<sbrk/brk error>: out of memory

הגדרות והצעות למימוש

בחלק זה של הנקודות מה שכתוב תחת הגדרות למימוש הוא חובה, ומה שכתוב תחת הצעות למימוש הוא הצעה וניתן להתעלם ממנו במידה הצורך.

הגדרות למימוש:

1. במידה ואתם בוחרים לשמר metadata לכל בלוק, יש לוודא שהגודל המוקצה לבלוק עצמו שווה לגודל שביקש המשתמש – המשתמש לא צריך "לשלם" בבלוק שלו את גודל header שבחרתם להכניס למימוש.
2. על כל הקצאות הזכורן להיות בגודל שהוא כפולה של 4 בתים. למשל, אם משתמש מבקש הקצהה של size = 5 בתים, יש להקצת בלוק של 8 בתים ולתת למשתמש. אם $0 \neq !size$, אין צורך "לחסום" את הבתים שהמשתמש לא ביקש (בחלק של הבלוק שגודול $size$), אלא להציג לו את המצביע לתחילת הבלוק בצורה רגילה. זה אפשרי לכם לשמר על גודל הקצהה כללי קל למעקב. ניתן להשתמש ב-macro הבא, שLAGUCH מספר שלהם כלשהו ומוגבל אותו למספר המינימלי שהוא כפולה של 4 וגודלו שווה לו:

```
#define ALIGN_TO_MULT_OF_4(x) ((4 + (2 << ((x) - 1) / 2)) * (x))
```

3. לתרגיל נתון שلد לשימושכם. אתם רשאים לשנות אותו בכל דרך פרט למסק **הfonקציות הנדרשות**. הfonקציות שתתמכשו יקרוו כמו שהן מופיעות ב-h.h,customAllocator, ובמידה וישנוו, לא יתקמפלו ולא יקבלו נקודות.
4. יש לוודא כי במידה ובזמן הקצתה זכרו קיים בлок פניו שנitin להקצתה, הוא הבלוק שיוקצה בהכרח ובשם סיטואציה/מקרה קצה לא יוקצה זכרו נוספת.
5. יש לוודא כי במידה ובזמן שחרור זכרו קיימת האפשרות לאיחוד בлокים, זה יתרבצע בהכרח לפני יציאה מהfonקציה free ללא יוצאים מן הכלל.
6. יש לוודא כי כל גודלי הזיכרון בתרגיל מיוצגים ע"י הסוג `t_size` ולא ע"י `int/unsigned long` וכו'.
7. לכל הfonקציות הנדרשות למימוש קיימת תחילתית של `custom` על מנת להמנע מקונפליקטים עם הfonקציות המקוריות ב-h.h.stdlib. אין מניעה לבצע `include` ל-h.h.stdlib במידת הצורך, אבל למען הסר ספק אין להשתמש ב-`free` ו/or אנטים של `malloc` ממנו.
8. רק עברו חלק א' – ניתן להניח שהתוכנית לא נקראת בצורה מקובלית ושחווט ייחיד מאותו תהליך ניגש בכל זמן נתון לכל וריאנט של `malloc`.
9. ניתן לשנות את הקובץ customAllocator.c ל-`customAllocator.cpp` על מנת כתוב ב-`C++`.
10. יש להגיש רק את הקבצים h.customAllocator.h וcustomAllocator.cpp. לתרגיל לא מסופק Makefile ואין צורך להגיש אחד – רצוי לכתוב קובץ cpp main.c/c/main.c ביחד עם Makefile משלכם על מנת לבדוק את המימוש ולכתוב טסטים שונים שיכולים לבדוק את המערכת בסיטואציות קצה.
11. אין צורך להסביר את הסוגרים המשולשים "<>" בהדפסות כמו בתרגילים הקודמים.

הצעות למימוש מבנה התוכנית:

1. במידה ותרצו ניתן להשתמש בfonקציות `heapCreate/Kill` שייקראו בתחילת סוף הבדיקה בשביל לאתחול משתנים וכו'. ניתן להשאיר אותן ריקות אבל אין לשנות את החתימה שלהן/שםן.
2. כדאי להגדיר חלק מהאובייקטים בתרגיל כסטטיים על מנת שיוקטו בתחילת ריצת התוכנית וישמרו לאורכה (כלומר, לא ישוחררו לאחר יציאה מה-`scope` של הfonקציות הנקראות).
3. כדאי להשתמש בfonקציית `memcpy` לצרכי העתקות פשוטות של בתים בין מקומות בזיכרון.
4. קראו את ה-`page` man של הfonקציות הנדרשות למימוש, אבל שימוש לב שחקן מהדרישות שונו על מנת להקל על המימוש בתרגיל.

ח'לק ב' – multithreaded malloc

נשים לב שהשימוש שכתבנו לעיל לא מתאים לנסיבות מרובות חוטים. קריאות לא מסונכרנות ל-`brk`-`sbrk` יכולות להוביל להקצאות ושחרורים לא חוקיים – תיכנו סיטואציות בהן חוט 1 מקבל את הזכרון שיעוד לחוט 2, או חוט 1 משחרר זכרון שחותט 2 עוד לא סיים אותו וכו'. יש כמה דרכים לפתרור בעיה זו:

1. הגדרת `malloc` כקטע קרייטי – זה אכן יוכל לעבוד, אבל יוכל לייצר צוואר בזיכרון משמעוני בתוכניות שצורך ומשחררות הרבה זכרון דינמי.
2. הגדרת `heap` לכל חוט – מקטין בהרבה את ה-`contention` על הקצת זכרון, אבל גורם לפרגמננטציה ומסובך מאוד בסיטואציות בהן מספר החוטים בתוכנית לא קבוע לאור התוכנית.
3. נעה עדינה – חלוקת `heap` לחלקים והגדרת מנעל לכל חלק. מצליח ביחס לאופציה 1, אבל מסובך לימוש ולדיבוג.
4. שילוב של גישות 3-2 – יצירת `heap` קטן לכל חוט בו הקצאות קטנות יקרו ו-`heap` גדול בו הקצאות גדולות יקרו.

הדרך המקובלת למימושים מסוימים של `malloc` היא גישה 4. עם זאת, נתאים את הפונקציות שכתבתם בשאר התרגיל לנסיבות מרובות חוטים ע"י גישה 3:

1. יש להתאים את ארבעת הפונקציות מהחלק הראשי של התרגיל וליצור להן גרסאות המתאימות לנסיבות מרובות חוטים – הן `customMTMalloc` ו-`customMTFree`.
2. יש לכתוב פונקציות `void heapCreate()` ו-`void heapKill()`. הפונקציות יקראו מיד עם תחילת ה-`main` וכייד לפני סוףו בהתאם בבדיקה, ויש להשתמש בהן על מנת לאותחל ולשחרר בהתאם את איזורי הזיכרון שיישמשו להקצתה.
3. יש להזכיר מראש 8 איזורי זיכרון שונים בגודל 4KB כל אחד, כל אחד עם מנעל מיוחד, המוכנים למעבר למשתמש (כלומר – לאחר `heapCreate()` עליהם להיות כבר מוקצים ומוכנים). יש לוודא כי קבלת ושחרור זיכרון מאותו איזור מתחבצת בצורה בטוחה במידה וחוטים רבים מנוסים לגשת אליו.
4. במידה ומתאפשרת בקשה להקצת זיכרון שגודלה מזה זיכרון הפנוי באיזור הבא שעליו לטפל בהקצתה (למשל – יש 1KB פנוי באיזור 3 והוא מקבל בקשה ל-2KB) יש להקצות איזור נוסף ונוסף ולהוסיף לתור האיזורים.
5. ניתן להניח שלא תתקבל בקשה להקצתה שגדולה מ-8KB (כלומר – אין סיטואציה בה יש להקצת יותר מאייזור אחד עavor בקשה אחת).
6. יש לוודא ההקצאות מתחבצות לאורק האיזורים בצורה של תור מעגלי. כל הקצתה מתחבצת בזכרון אחר, עד שכל איזורי זיכרון יושמו לפחות פעם אחת, ורק אז להשתמש באיזור זיכרון פעם נוספת. למשל, אם יש 8 איזורים ו-8 חוטים ביצעו כל אחד הקצתה אחת, כל אחד מהאיזורים יתן מענה לחוט אחר, ללא `contention`. בהקצתה התשיעית איזור מס' 0 יתן מענה, לאחר מכן איזור 1 בהקצתה העשירית, וכך הלאה. ניתן לדלג בסבב זהה על איזורים שאינם מספק מקום למתנה לבקשת הנוכחית.
7. ח'לק ב' יבדקivid ביחיד ובנפרד מחלק א' – הם יקומפלו וירצנו ייחדיו. יש לוודא שגםם לא מפריעים זה לזה מבחינת שמות משתנים, הצהרות פונקציות וכו'.
8. אין להשתמש ב-`thread local storage` (המזהה `__thread`).

חלק ג' – טסיטים ופרומפטים של AI

1. מומלץ אך לא הכרחי להשתמש בכלים AI לכתיבת קוד.
2. חשוב שתקרו ותבינו במדוק מה הקוד עשו ותשנו אותו בהתאם לצורך ובהתאם לדרישות התרגיל. שימו לב, הדרך הטובה ביותר ללמידה זה לנוטות לכתוב את הגרסת ההתחלתית לבד ואז להשתמש באי.
3. לא כדאי להעלות את התרגיל ולבקש לפתור אותו, אלא לבקש לכתוב רשימת דרישות ההתחלתית שאתם מבינים אותה ומודדים שהיא מלאה ונכונה, ורק אז ליצר קוד באופן הדרגתי ומובקר.
4. עלייכם לספק log של כל ההסיאנס שלכם מול כלי ה-AI. ה-log יכול את הקלטים שלכם ואת הפלטים של כלי ה-AI הן מלול והן קבצים.
5. כל קובץ log יוכל במשפט הראשון שלו את שם הכליל וגרסת המודול מоловו עבדתם. אם עבדתם עם יותר מכלי 1 יש להוסיף לוג עבור כל הכליל. שם הקוד הוא-`index.txt`.
6. הפרומפטים חייבים להיות מלאים ולהכיל את כל ההתקנתביות.
7. עלייכם ליצור ולהריץ טסיטים מדוייקים שבודקים בצורה מפורשת היבטים מסוימים של המימוש, מוביל להסתמך על AI עצמו כדי ליצור אותם. תפרטו בבדיקה מה התקנות שאתם בודקים, גם אם משתמשים ב AI וגם אם לא. עלייכם לספק את הקוד של כל הטסיטים לבדיקה שלנו.

הגשה, קומפילציה וLINKAGE

יש להגיש את הקבצים הבאים (והקבצים הבאים בלבד):

- .1 customAllocator.h
- .2 customAllocator.cpp **ו** customAllocator.c

בבדיקה שני הקבצים האלה יורדו לתיקיה ביחד עם קובץ בשם main.c או main.cpp ויקומפלע ע"י הפקודות הבאות בהתאם:

```
> g++ -std=c++11 -Wall -Werror -pedantic-errors -DNDEBUG *.cpp -o main  
> gcc -std=c99 -Wall -Werror -pedantic-errors -DNDEBUG *.c -o main
```

יש להוסיף את הדגל `-pthread` לקומpile.

הדגל `Werror` גורם ל-`warnings` רגילים בקומpileר `gcc/g++` להפוך ל-`errors`. יש לוודא שהקוד שלנו מתקmpl על המcona של הקורס ולא `warnings/errors`.

קוד שלא יתкомפל/יתקmpl עם אזהרות לא יקבל ציון.

מומלץ להפריד את המימוש לקבצי `c/cpp`.`.h`. על מנת להקל על בניית התוכנית.

בצלחה!