תרגיל רטוב #3

ניהול הקצאת זכרון דינמית

הקדמה

שימו לב – מקוריות הקוד תבדק. העתקת שיעורי בית היא עבירת משמעת בטכניון על כל המשתמע מכך.

בתרגיל זה נממש גרסא משלנו לפונקציות המוכרות malloc, free ואחרות מ-stdlib.h על מנת ללמוד בצורה מעמיקה יותר על מרחב הזכרון של תהליך, מנגנוני הקצאת זכרון ומעקב אחריהם, יעילות ההקצאה, תקשורת עם מערכת ההפעלה ועוד.

כמה טיפים:

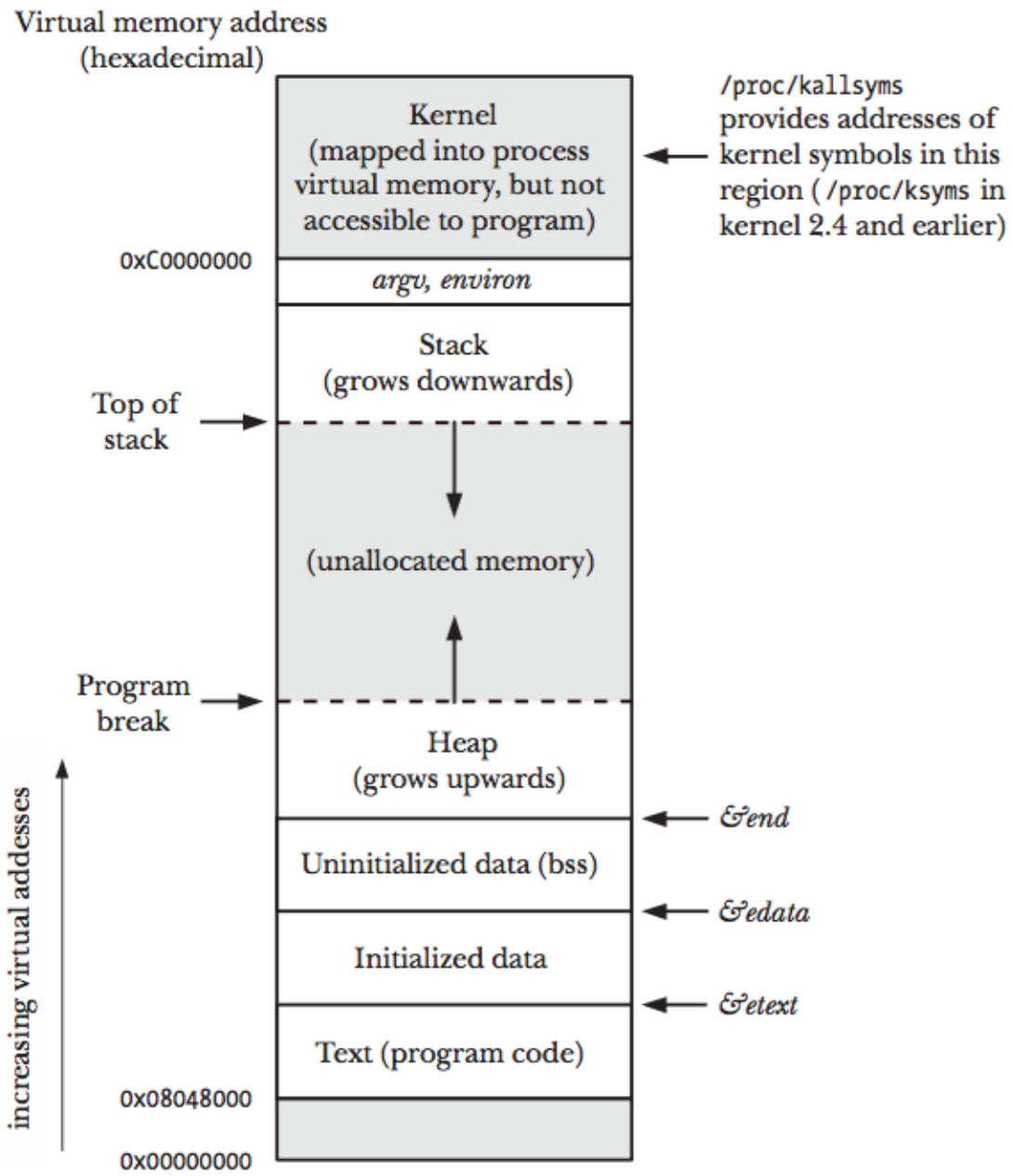
1. לפני ששואלים שאלות בפורום – עשו מאמץ לוודא שהתשובה לשאלה שלכם לא כתובה בהנחיות התרגיל/נשאלה כבר בפורום/בקבוצה של הקורס. דגש מיוחד לתרגיל הזה – שאלות כמו "האם זה בסדר שנעלתי את X בזמן ש-Y..." הן חלק ממימוש הנדרש בתרגיל.
2. בחלק הבא של המסמך תמצאו רקע תיאורטי לתרגיל ולאחריו ההנחיות המדויקות. קראו קודם כל את רקע והעזרו בתרגול על malloc על מנת לתכנן ורק לאחר מכן התחילו לכתוב קוד.
3. **תכננו קודם!** שרטטו תרשים זרימה כללי של התוכנית. מה הם מבני הנתונים הנדרשים? איך נממש אותם? הגדירו הצהרות של classes/structs, תכננו פונקציות שתרצו לממש ורק אח"כ תתחילו לכתוב. ברגע שמתחייבים למימוש מסוים שלא לוקח בחשבון את אחת הדרישות, קשה הרבה יותר לשנות.
4. למדו להשתמש ב-gdb – בקורס ממ"ת יש סדנא עם כל מה שצריך לדעת על gdb, וזה כלי מאוד שימושי להבין מה מצב של כל משתנה בכל זמן, להשתמש ב-breakpoints וכו'.

בהצלחה!

רקע תיאורטי

הערה – החומר הזה מופיע בפירוט בתרגול 8 (זכרון וירטואלי #2).

מבנה הזכרון הוירטואלי של תהליך בלינוקס נראה סכמטית כך:



אנחנו נתעניין ספציפית ב-heap. הקצה העליון של ה-heap נקרא program break ה- של התהליך והוא יכול לגדול ולקטון בצורה דינמית בהתאם לדרישות התהליך. ה-stack גם יכול לגדול ולקטון בצורה דינמית, אבל בדרך כלל מנוהל ללא מעורבות המשתמש לפני ריצת התוכנית ע"י הקומפיילר (קיפול ודחיפה של המחסנית בקריאה לפונקציות למשל). ה-heap, לעומתו, מנוהל ידנית ודינמית ע"י קריאות המערכת brk/sbrk שמבצע המשתמש במהלך ריצת התוכנית.

במהלך ריצה של התהליך ה-heap גדל כלפי מעלה בהקצאת זכרון ע"י פונקציות כמו malloc, וזה מתבטא בהגדלה של program break ה-. עם זאת, לא כל קריאה ל-free יכולה להקטין בהכרח את ה- program

break – למשל, אם נשחרר זכרון שנמצא מתחת לזכרון אחר שעדיין מוקצה, לא נוכל להקטין את ה-program break ותיווצר פרגמנטציה. עם זאת, כן נוכל להקצות את הזכרון ששוחרר בפעם הבאה שהתהליך קורא ל-malloc.

קריאות מערכת רלוונטיות:

קריאות המערכת הבאות ישמשו אותנו בכתיבת התרגיל:

```
int brk(void* addr);

void* sbrk(size_t increment);
```

יש לקרוא את ה-man pages שלהן לעומק לקראת מימוש התרגיל – שימו לב לפונקציונליות שלהם, ערכי החזרה, קביעת ערכים עבור errno וכו'.

פרטים נוספים למימוש:

נרצה לייצר מנגנון לניהול ושחרור דינמי של זכרון ע"י הפונקציות malloc (והוריאציות שלה) ו-free. בתרגיל נממש מנגנון פשוט יותר מה-buddy system להקצאת ושחרור בלוקים, שיפעל כך:

1. התוכנית תחזיק רשימה מקושרת של בלוקים שהוקצו.
2. לפני שבוצעו הקצאות, הרשימה תהיה ריקה.
3. כאשר מתבצעת הקצאת זכרון ע"י malloc או וריאנטים שלה:
   1. ראשית יש לנסות להחזיר בלוק קיים וחופשי כלשהו שכבר הוקצה ונמצא מתחת ל-program break. רק במידה ולא קיים בלוק מספיק גדול, יש להגדיל את מרחב הזכרון הדינמי של התהליך.
   2. אם מקצים בלוק קיים, יש לבחור את הבלוק להקצאה בשיטת ה-best fit – כלומר, הבלוק שגודלו גדול/שווה לגודל הרצוי אבל הכי קרוב לגודל הרצוי. למשל אם נדרש להקצות 1KB ויש בלוקים 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 בלוקים חופשיים סמוכים, יש לאחד אותם לבלוק אחד גדול יותר.
   2. אם ניתן לשחרר זכרון למערכת ההפעלה (כלומר, הבלוק בקצה מרחב הזכרון), יש לבצע זאת כחלק מהפונקציה free ולשחרר את מקסימום הזכרון האפשרי.

דרישות התרגיל

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

:void* customMalloc(size_t size)

1. מקצה size בתים ומחזירה מצביע לתחילת הזכרון המוקצה.
2. יש לוודא כי הפונקציה לא מקצה זכרון כשאין צורך לכך (יש מספיק זכרון רציף ופנוי מתחת ל-program break).
3. הפונקציה תבחר בלוק פנוי בשיטת ה-best fit.
4. הפונקציה לא תבצע שום פעולה על הזכרון שניתן לה לפני שתחזיר אותו למשתמש.
5. במידה ו-0 => size, הפונקציה תחזיר NULL ותדפיס ל-stderr:

```
<malloc error>: passed nonpositive size
```

:void customFree(void* ptr)

1. הפונקציה תקבל מצביע שהתקבל ע"י וריאנט של malloc ותסמן את הזכרון ששייך לו כפנוי.
2. במידת האפשר, הפונקציה תאחד בלוקים.
3. במידת האפשר, לאחר השחרור הפונקציה תקטין את ה-program break של התהליך ככל שניתן.
4. אם ptr הוא NULL, הפונקציה לא תבצע שום פעולה הקשורה לזכרון ותדפיס שגיאה:

```
<free error>: passed null pointer
```

5. ניתן להניח שכל הקריאות לפונקציה מתבצעות עם מצביע שהוא או NULL או הוחזר מוריאנט של malloc, ולא מצביע לזכרון במקום אחר (למשל על ה-stack).

הערה – דרישה 5 היא דרישה מקלה לטובת התרגיל, במימוש המלא כפי שמופיע ב-stdlib.h קיים טיפול במקרה זה שלרוב מייצר interrupt מסוג abort ע"י מעקב אחרי טווח הכתובות המוקצות.

:void* customCalloc(size_t nmemb, size_t size)

1. מקצה nmemb אלמנטים שגודל כל אחד מהם בגודל size בתים, לסה"כ nmemb * size בתים, ותכתוב את הערך 0 לכל אחד מהבתים הללו.
6. יש לוודא כי הפונקציה לא מקצה זכרון כשאין צורך לכך (יש מספיק זכרון רציף ופנוי מתחת ל-program break).
7. במידה ו-0 => nmemb או 0 => size, הפונקציה תחזיר NULL ותדפיס ל-stderr:

```
<calloc error>: passed nonpositive value
```

המשך בעמוד הבא.

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

1. הפונקציה מקבלת מצביע ptr שהוחזר ע"י וריאנט של malloc וגודל size בבתים. נסמן ב-old_size את הגודל שהוקצה ל-ptr במקור. מטרת הפונקציה היא לשנות את הגודל של הבלוק הזכרון אליו ptr מצביע, בהתאם להגדרות הבאות.
2. אם size < old_size, המטרה היא לשחרר זכרון (אם אפשר). במידת האפשר בהתאם לקונפיגורציית הבלוקים והגדלים, הפונקציה תנסה לשחרר את הזכרון שבין size ל-old_size. אם הפונקציה לא מצליחה לשחרר את הבלוקים בצורה המתאימה, יתבצעו הפעולות הבאות:
    1. הפונקציה תקצה בלוק חדש בגודל size (הקטן מה-old_size)
    2. תעתיק לבלוק החדש את תכני הבלוק הישן שבין תחילתו לבין size בתים (כלומר, תשמיט מהבלוק החדש את החלק "שמעל" ל-size)
    3. תשחרר את הבלוק הישן
    4. תחזיר מצביע לבלוק החדש
3. אם size > old_size:
    1. הפונקציה תקצה בלוק חדש בגודל size
    2. תעתיק לבלוק החדש את כל תכני הבלוק הישן
    3. תשחרר את הבלוק הישן
    4. תחזיר מצביע לבלוק החדש
4. אם ptr == NULL, הפונקציה תהיה שקולה ל-malloc(size).
5. ניתן להניח שכל הקריאות לפונקציה מתבצעות עם מצביע שהוא או NULL או הוחזר מוריאנט של malloc, ולא מצביע לזכרון במקום אחר (למשל על ה-stack).

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

```
<sbrk/brk error>: out of memory
```

הגדרות והצעות למימוש

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

הגדרות למימוש:

1. על כל הקצאות הזכרון להיות בגודל שהוא כפולה של 4 בתים. למשל, אם המשתמש מבקש הקצאה של size = 5 בתים, יש להקצות בלוק של 8 בתים ולתת למשתמש. אם size % 4 != 0, אין צורך "לחסום" את הבתים שהמשתמש לא ביקש (בחלק של הבלוק שגדול מ-size), אלא להחזיר לו את המצביע לתחילת הבלוק בצורה רגילה. זה יאפשר לכם לשמור על גודל הקצאה כללי קל למעקב.
ניתן להשתמש ב-macro הבא, שלוקח מספר שלהם כלשהו ומעגל אותו למספר המינימלי שהוא כפולה של 4 וגדול/שווה לו:

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

2. לתרגיל נתון שלד לשימושכם. אתם רשאים לשנות אותו בכל דרך **פרט לממשק הפונקציות הנדרשות**. הפונקציות שתממשו יקראו כמו שהן מופיעות ב-customAllocator.h, ובמידה וישונו, לא יתקמפלו ולא יקבלו נקודות.
3. יש לוודא כי במידה ובזמן הקצאת זכרון קיים בלוק פנוי שניתן להקצות, הוא הבלוק שיוקצה בהכרח ובשום סיטואציה/מקרה קצה לא יוקצה זכרון נוסף.
4. יש לוודא כי במידה ובזמן שחרור זכרון קיימת האפשרות לאיחוד בלוקים, זה יתבצע בהכרח לפני יציאה מהפונקציה free ללא יוצאים מן הכלל.
5. יש לוודא כי כל גדלי הזכרון בתרגיל מיוצגים ע"י הסוג size_t ולא ע"י int/unsigned/long וכו'.
6. לכל הפונקציות הנדרשות למימוש קיימת תחילית של custom על מנת להמנע מקונפליקטים עם הפונקציות המקוריות ב-stdlib.h. אין מניעה לבצע include ל-stdlib.h במידת הצורך, אבל **למען הסר ספק אין להשתמש ב-free ווריאנטים של malloc ממנו**.
7. במידה ואינכם מגישים את תרגיל הבונוס – ניתן להניח שהתוכנית לא נקראת בצורה מקבילית ושחוט יחיד יחיד מאותו תהליך ניגש בכל זמן נתון לכל וריאנט של malloc.
8. ניתן לשנות את הקובץ customAllocator.c ל-customAllocator.cpp על מנת לכתוב ב-C++.
9. יש להגיש רק את הקבצים customAllocator.h ו-customAllocator.c/cpp. לתרגיל לא מסופק Makefile ואין צורך להגיש אחד – רצוי לכתוב קובץ main.c/cpp ביחד עם Makefile משלכם על מנת לבחון את המימוש ולכתוב טסטים שונים שיכולים לבחון את המערכת בסיטואציות קצה.
10. אין צורך להסיר את הסוגריים המשולשים "<>" בהדפסות כמו בתרגילים הקודמים.

הצעות למימוש מבנה התוכנית:

1. כדאי להגדיר חלק מהאובייקטים בתרגיל כסטטיים על מנת שיוקצו בתחילת ריצת התוכנית וישמרו לאורכה (כלומר, לא ישוחררו לאחר יציאה מה-scope של הפונקציות הנקראות).
2. כדאי להשתמש בפונקציה memcpy לצרכי העתקות פשוטות של בתים בין מקומות בזכרון.
3. קראו את ה-man page של הפונקציות הנדרשות למימוש, אבל **שימו לב שחלק מהדרישות שונו על מנת להקל על המימוש בתרגיל**.

תרגיל בונוס – multithreaded malloc

על החלקים בתרגיל שהוגדרו למעלה ניתן לקבל לכל היותר 100 נקודות. על תרגיל הבונוס שיפורט כעת ניתן לקבל 25 נקודות כך שהציון המקסימלי בתרגיל הוא 125 – משקל התרגיל הוא 4 נקודות ולכן הציון המקסימלי שניתן להרוויח על התרגיל הוא 5 = 4 * 1.25, כלומר ניתן להרוויח נקודה נוספת לציון הסופי.

נשים לב שהמימוש שכתבנו לעיל לא מתאים לסביבה מרובת חוטים. קריאות לא מסונכרנות ל-brk ו-sbrk יכולות להוביל להקצאות ושחרורים לא חוקיים – יתכנו סיטואציות בהן חוט 1 מקבל את הזכרון שיועד לחוט 2, או שחוט 1 משחרר זכרון שחוט 2 עוד לא סיים איתו וכו'. יש כמה דרכים לפתור בעיה זו:

1. הגדרת malloc כקטע קריטי – זה אכן יכול לעבוד, אבל יכול לייצר צוואר בקבוק משמעותי בתוכניות שצורכות ומשחררות הרבה זכרון דינמי.
2. הגדרת heap לכל חוט – מקטין בהרבה את ה-contention על הקצאת זכרון, אבל גורם לפרגמנטציה ומסובך מאוד בסיטואציות בהן מספר החוטים בתוכנית לא קבוע לאורך התוכנית.
3. נעילה עדינה – חלוקת ה-heap לחלקים והגדרת מנעול לכל חלק. מוצלח ביחס לאופציה 1, אבל מסובך למימוש ולדיבוג.
4. שילוב של גישות 2-3 – יצירת heap קטן לכל חוט בו הקצאות קטנות יקרו ו-heap גדול בו הקצאות גדולות יקרו.

ישנם פתרונות נוספים, אבל דרך מקובלת לטפל בבעיה הזו (המבוצעת במימושים פורמליים של glibc או של חברות פרטיות כמו tcmalloc של גוגל) היא גישה 4.

תפקידכם בתרגיל הוא להתאים את הפונקציות שכתבתם בתרגיל לסביבה מרובת חוטים לפי גישה 3,:

1. יש להתאים את ארבעת הפונקציות מהחלק הראשי של התרגיל וליצר להן גרסאות המתאימות לסביבה מרובת חוטים – הן נקראות customMTMalloc ונמצאות בשלד בהערה אותו יש להסיר במידה ואתם מעוניינים להגיש את הבונוס.
2. יש לכתוב פונקציות void heapCreate() ו-void heapKill(). הפונקציות יקראו מיד עם תחילת ה-main ומיד לפני סופו בהתאמה בבדיקה, ויש להשתמש בהן על מנת לאתחל ולשחרר בהתאמה את איזורי הזכרון שישמשו להקצאה.
3. יש להכין מראש 8 איזורי זכרון שונים בגודל 4KB כל אחד, כל אחד עם מנעול משלו, המוכנים למעבר למשתמש (כלומר – לאחר heapCreate() עליהם להיות כבר מוקצים ומוכנים). יש לוודא כי קבלת ושחרור זכרון מאותו איזור מתבצעת בצורה בטוחה במידה וחוטים מרובים מנסים לגשת אליו.
4. במידה ומתקבלת בקשה להקצאת זכרון שגדולה מהזכרון הפנוי באיזור הבא שעליו לטפל בהקצאה (למשל – יש 1KB פנוי באיזור 3 והוא מקבל בקשה ל-2KB) יש להקצות איזור נוסף ולהוסיפו לתור האיזורים.
5. ניתן להניח שלא תתקבל בקשה להקצאה שגדולה מ-4KB (כלומר – אין סיטואציה בה יש להקצות יותר מאיזור אחד עבור בקשה אחת).
6. יש לוודא ההקצאות מתבצעות לאורך האיזורים בצורה של תור מעגלי. כל הקצאה מתבצעת בזכרון אחר, עד שכל איזורי הזכרון שומשו לפחות פעם אחת, ורק אז להשתמש באיזור זכרון פעם נוספת. למשל, אם יש 8 איזורים ו-8 חוטים ביצעו כל אחד הקצאה אחת, כל אחד מהאיזורים יתן מענה לחוט אחר, ללא contention. בהקצאה התשיעית איזור מספר 0 יתן מענה, לאחר מכן איזור 1 בהקצאה העשירית, וכך הלאה. ניתן לדלג בסבב הזה על איזורים שאין בהם מספיק מקום לתת מענה לבקשה הנוכחית.
7. תרגיל הבונוס יבדק ביחד עם התרגיל הראשי – הם יקומפלו ויורצו יחדיו. יש לוודא ששניהם לא מפריעים זה לזה מבחינת שמות משתנים, הצהרות פונקציות וכו'.
8. אין להשתמש ב-thread local storage (המזהה __thread).

הגשה, קומפילציה ולינקוג'

יש להגיש את הקבצים הבאים (והקבצים הבאים בלבד):

1. קובץ readme בהתאם להנחיות בנוהל תרגילי הבית
2. customAllocator.h
3. customAllocator.c **או** customAllocator.cpp

בבדיקה שלושת הקבצים האלה יורדו לתיקייה ביחד עם קובץ בשם main.c או main.cpp ויקומפל ע"י הפקודות הבאות בהתאמה:

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

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

הדגל Werror גורם ל-warnings רגילות בקומפיילר gcc/g++ להפוך ל-errors. יש לוודא שהקוד שלכם מתקמפל **על המכונה של הקורס** ללא warnings/errors.

קוד שלא יתקמפל/יתקמפל עם אזהרות לא יקבל ציון.

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

בהצלחה!