מוקד כרמיאל

תיכון אמירים
כפר ורדים

Bug OS

מערכת הפעלה של 32Bit למחשבים מבוססי ארכיטקטורת Intel x86

30.4.2016

אבידן בוריסוב (ת.ז. 318740347)

תיכון אמירים כפר ורדים

מנטור – משה פינטו

אחראי פרוייקט – יואב פורשטיין

תוכן עניינים

רקע..... 2
תיאור שיווקי של המוצר..... 3
סביבת העבודה 5
אלגוריתמים מרכזיים בפרויקט..... 6
ממשק משתמש 9
רפלקציה אישית..... 12
המוצר המוגמר אל מול התכנון הראשוני 14
ביבליוגרפיה..... 15

רקע

Bug OS היא מערכת הפעלה חדשה וייחודית ב-32Bit למחשבים מבוססי ארכיטקטורת Intel x86. המערכת נכתבה לגמרי מההתחלה והיא אינה מבוססת על אף מערכת הפעלה קיימת בשוק. מטרת הפרויקט היא ליצור מערכת הפעלה טובה, שימושית ובעלת כל המאפיינים שמשתמש מצפה למצוא במערכת הפעלה.

הסיבה העיקרית שבחרתי בפרויקט זה היא על מנת להבין בצורה מלאה כיצד פועלות מערכות ההפעלה מאחורי הקלעים, לדעת מה המנגנונים ומה מתרחש מאחורי הקלעים בכל פעולה שתוכנות מחשב מבצעות. מטרתי הייתה למעשה לדעת לממש בעצמי את כל הקוד שרץ על המחשב וליצור מערכת עבור המשתמש שלא תלויה באף גורם אחר – מערכת אשר היא האחראית לכל הנעשה המחשב. במיוחד סקרן אותי לדעת כיצד מתרחש ריבוי תהליכים עם זיכרון וירטואלי, וכיצד למעשה מתנהל כל מנגנוני הקלט והפלט מחומרה חיצונית שמערכת ההפעלה אחראית עליו.

המוצר אינו מבוסס על אף טכנולוגיה קיימת בשוק. המוצר נכתב בשפות -x86 Assembly וב-C++, תוך דגש על עיצוב מודרני (בפרט נעשה שימוש ב-C++14), פשוט, קריא ותחזוקתי.

מערכת ההפעלה מורכבת מ-3 חלקים מרכזיים: ה-bootloader, ה-kernel וה-user space. בנוסף, המערכת מורכבת מ-2 ספריות שמשמשות את חלקיה המרכזיים - הספרייה הסטנדרטית (stdlib) שמכילה API מקביל לספרייה הסטנדרטית של C++ שמשמשת גם את ה kernel וגם את ה- user space, וספריית המשתמש (userlib) שמטרתה לספק API לתקשור עם המערכת לתוכנות משתמש.

ה-bootloader הוא הרכיב הראשוני שרץ ותפקידו רק לטעון את ה-kernel לזיכרון, ולאחר מכן להעביר את השליטה ל-kernel.

ה-kernel הוא הרכיב המרכזי במערכת ההפעלה שמורכב מהמון רכיבים נפרדים פנימיים שתפקידם לשלוט על תפקוד כל חלק במחשב. ה-kernel, בנוסף להיותו הרכיב שאחראי לפעולה של כל מה שרץ במחשב, משמש כגשר ל-user space שמאפשר לו להתממשק עם רכיבים חיצוניים, ומספק לו שירותים שונים.

ה-user space הוא ה-front end של מערכת ההפעלה. רכיב זה כולל את כל התוכנות אשר עושות שימוש במערכת ההפעלה, בין אם תוכנות מערכת או תוכנות עבור המשתמש.

מערכת ההפעלה מהווה פתרון טוב למשתמש שמעוניין לקבל מערכת שתנהל את המחשב שהוא קנה. ללא מערכת הפעלה, אי אפשר למעשה להשתמש במחשב, לכן המוצר הוא הכרחי. הבעיה העיקרית בשימוש במערכת הפעלה זו היא שמדובר במערכת הפעלה חדשה ולכן אין לה תוכנות רבות שמשתמש יכול למצוא (בהשוואה למערכות הפעלה קיימות).

תיאור שיווקי של המוצר

המוצר הינו מערכת הפעלה כוללת ומלאה למחשבים בעלי ארכיטקטורה תואמת ל-Intel i386. (ארכיטקטורות חדשות יותר בעלי תמיכה לאחור, כגון אלו במעבדי אינטל מבוססי x86-64, נתמכות גם כן).

המוצר אינו מבוסס על שום מערכת הפעלה מוכרת בשוק. כולו נכתב מאפס – מהרגע שהמחשב נדלק ועד שהוא נכבה, הקוד היחידי שירוץ יהיה קוד שנכתב עבור מערכת ההפעלה.

מערכת ההפעלה ניתנת להפעלה מכל מחשב ביתי מצוי מתוך DOK בצורה חייה (Live-USB).

מערכת ההפעלה לא משתמשת ב-bootloader חיצוני – היא מכילה bootloader ייעודי שמצוי ב-boot sector של ה-USB שיודע למצוא ולהעלות את הליבה.

מערכת ההפעלה יודעת להתממשק עם רכיבים חיצוניים כגון התקני אכסון, מסך, מקלדת, שעון.

מערכת ההפעלה פועלת בצורה מאובטחת (היא רצה ב-Protected Mode עם Paging), לכל תהליך מובטחות הגנות שונות ומרחב זיכרון וירטואלי פרטי. תהליכי משתמש לא יכולים לגשת למרחב הזיכרון של ה-kernel וגם לתהליכים אחרים. תהליך משתמש לא יכול לחדור ל-kernel, הוא מסוגל רק לבצע קריאות מערכת ל-kernel (תהליך שמבצע פקודה לא חוקית ייסגר).

שימוש במוצר:

- מערכת ההפעלה יודעת להריץ תהליכים אשר המשתמש יתקשר איתם.
- התהליכים רצים במקביל
- מערכת ההפעלה מספקת שירותים בסיסיים לתוכנות שרצות עליה (קריאות מערכת, ספריות).
- מערכת ההפעלה מספקת תוכנות עבור המשתמש.
- המשתמש מריץ תוכנות ופקודות שונות דרך ה-shell. בעזרתו המשתמש יכול לבצע משימות ניהול כגון כתיבה וקריאה של קבצים, הצגת הקבצים והתיקיות במערכת, ניווט במערכת, קבלת מידע על התהליכים שרצים, ניהול תהליכים, קבלת מידע על הזיכרון בשימוש, לחכות לפרק זמן מסויים, להריץ תוכנות נוספות (כגון מחשבון ומשחקים) ועוד.
- ניתן לשחק בסנייק על מנת להעביר את הזמן ☺

משתמש שרוצה מערכת הפעלה למחשב שלו ירצה לשלם כסף עבור המוצר משום שללא מערכת הפעלה הוא לא יוכל להשתמש במחשב. באמצעות מערכת ההפעלה, המשתמש יוכל לבצע את המשימות שהוא צריך ממחשב – הוא יוכל לערוך קבצים, לשלוט על תהליכים ולפתוח תוכנות, לשחק משחקים, לחשב ביטויים מתמטיים ועוד.

בעל ממון ירצה להשקיע בפיתוח מערכת ההפעלה משום שעם כוח אדם גדול מספיק, Bug OS יכולה להפוך למערכת הפעלה שימושית ביותר עבור משתמשים רבים, ולנגוס בנתח השוק של מערכות הפעלה קיימות.

בהשוואה למערכות ההפעלה הקיימות היום, כגון Windows וLinux, Bug OS אינה מכילה יתרונות רבים. פיתוח מערכות הפעלה אלו נעשה במשך עשרות שנים ע"י חברות ענק עם אלפי עובדים במשרה מלאה וכוח אדם – בעוד ש Bug OS פותחה ע"י מפתח אחד. משתמשים יעדיפו להשתמש במערכות הפעלה קיימות בעיקר משום שרוב התוכנות בשוק מיועדות לרוץ עליהם ולא על מערכות הפעלה אחרות.

עם זאת Bug OS מספקת הזדמנות מצוינת למשתמשים שמתעניינים בדברים שמתרחשים מאחורי הקלעים של מערכת הפעלה, לקבל הצצה לקוד קריא וברור שמדגים כיצד מערכת הפעלה עובדת. בנוסף, Bug OS יכולה להיות שימושית בתור מערכת הפעלה למחשבים ישנים בעלי מעט זיכרון וכוח עיבוד חלש – היות שהמערכת אינה דורשת הרבה משאבים לעומת מערכות ההפעלה הקיימות בשוק.

תכנון עתידי - המערכת תכיל מערכת התממשקות גראפית (GUI) שתפעל באמצעות שימוש בעכבר.

סביבת העבודה

מערכת ההפעלה כתובה כולה ב-x86 Assembly וב-C++. ה-bootloader כתוב באסמבלי, הקרנל מורכב משילוב של C++ ואסמבלי, וה-user space נכתב ב-C++. הספרייה הסטנדרטית של מערכת ההפעלה כתובה ב-C++ וספריית ה-user space שמספקת מערכת ההפעלה כתובה ב-C++ ואסמבלי.

בחרתי להשתמש ב-C++ ואסמבלי בעיקר משום שאין יותר מידי אלטרנטיבות, אבל גם כי אלו השפות שאני הכי בקיא בהן. שפות דינאמיות, שפות סקריפט או שפות עם VM לא היו אופציה כלל מכיוון שעל מנת לייצר קוד שירוץ כמערכת ההפעלה יש צורך בקומפיילר שמייצר קוד נייטיבי בלבד. בחרתי ב-C++ על פני שפות נייטיביות אחרות (כגון C, Pascal, Rust) מפני שזאת שפה שאני יודע ברמה גבוהה מאוד וגם מפני שיש לה יכולות רבות מאוד, במיוחד בסטנדרט החדש שלה (C++14). השימוש באסמבלי לא היה מלווה בבחירה בכלל – מימוש מערכת הפעלה מחייב שימוש באסמבלי (של הארכיטקטורה שעליה רצה מערכת ההפעלה).

סביבת העבודה שבה השתמשתי לפיתות הפרוייקט היא Qt Creator. סביבה זו מיועדת לפיתוח תוכנות C++ רגילות (במיוחד כאלו שמשתמשות ב-Qt), אך הסביבה כה גמישה שהיא מצויינת גם לפיתוח מערכות הפעלה.

קמפול מערכת ההפעלה יכול להיעשות רק בלינוקס. תהליך הבנייה דורש קרוס קומפיילר של GCC 5.1.0 ו-QMake (מערכת בנייה שמגיעה עם Qt Creator).

מערכת ההפעלה יכולה לרוץ מתוך כל התקן שמכיל לפחות 1.44MB שה-BIOS מאפשר לטעון מתוכו boot sector, למשל: DOK, CD, דיסקט. בפועל מערכת ההפעלה מיועדת לרוץ מתוך כל DOK סטנדרטי (תוך שימוש ב-Floppy emulation).

בדיקת המערכת והרצתה נעשתה באמצעות שני אמולטורים ל-x86 – QEMU ו-BOCHS. השימוש בכלים אלו אפשר למעשה את הפיתוח של המערכת, שכן באמצעותם יכולתי להריץ כל פעם מערכת ההפעלה במחשב עליו עבדתי ללא צורך במחשב נפרד. בנוסף, אמולטורים אלו מגיעים עם GDB Server ייעודי – בו עשיתי שימוש נרחב על מנת לדבג את מערכת הפעלה.

על מנת להריץ את מערכת ההפעלה ב-QEMU או BOCHS, ניתן להשתמש בסקריפטים run-qemu.sh או run-bochs.sh.

על מנת לדבג את מערכת ההפעלה דרך Qt Creator, יש להריץ את debug-qemu.sh ואז לבחור באפשרות Debug -> Connect to GDB Server ב-Qt Creator ולבחור את קובץ האובייקט של הקרנל / bootloader / תוכנת משתמש.

אלגוריתמים מרכזיים בפרויקט

אלגוריתם להקצאת זיכרון דינאמי בסדר גודל של בתים (Heap)

הבעיה - בהינתן מנגנון להקצאת זיכרון בגודל גדול (סדר גודל של דפי זיכרון – 4KB), כיצד לממש מנגנון להקצאת זיכרון דינאמי של בתי זיכרון (במילים אחרות, מימוש של malloc() ו-free() של C)? יש להתחשב בכך שהאלגוריתם ימנע מיצירת "חורים" בזיכרון.

הפתרון הנבחר - שימוש ברשימות מקושרות מעגליות שמורכבות מה-Headerים שמכילים את המידע על הבלוק בזיכרון שנמצא מיד אחריו בזיכרון. לא ידוע לי על שם האלגוריתם, אבל הוא מתואר בספר The C Programming Language. הסבר כללי ניתן למצוא כאן.

תיאור האלגוריתם:

Header - מבנה נתונים שמכיל מצביע ל-Header הבא ואת גודל בלוק הזיכרון שאותו הוא מייצג. מיד אחרי סופו מופיע בלוק הזיכרון שמוחזר למשתמש. בתיאור האלגוריתם הכוונה בחוליה היא למצביע ל-Header, ובגודל החוליה הכוונה לגודל שאותו מייצג ה-Header שאליו מצביעה החוליה. הכוונה בחוליה->הבא היא לחוליה הבאה שהחוליה הנוכחית מצביע עליה.

בהתחלה יש Header סטטי אחד שהחוליה הבאה שהוא מצביע עליה היא עצמו, וגודלו 0 (לאחריו לא מוקצה זיכרון). יש מצביע שמצביע על ה-Header הזה והוא משמש כראש הרשימה המקושרת המעגלית (שמו "הראש"). הרשימה תתמלא במשך הזמן להיות "הרשימה של החוליות הפנויות".

פונקציית הקצאת הזיכרון – מקבלת מספר בתים שיש להקצות ומחזירה מצביע לבלוק זיכרון מוקצה:

1. מוסיפים למספר הבתים שהמשתמש מבקש את הגודל של Header אחד ומעגלים מספר זה לכפולה שלמה של הגודל של Header. נסמן גודל זה כ-X.
2. יש 2 מצביעי חוליות – נוכחית וקודמת. מאתחלים אותם כך שהחוליה הנוכחית הראש->הבא והחוליה הקודמת היא הראש.
3. לולאה אינסופית:
    1. אם גודל החוליה הנוכחית גדול או שווה ל-X:
        i. אם גודל החוליה הנוכחית שווה ל-X:
            1. הוצא את החוליה הנוכחית מהרשימה (גרום לקודמת->הבא להצביע על נוכחית->הבא).
        ii. אחרת:
            1. החסר מגודל החוליה הנוכחית X בתים.
            2. קדם את המצביע 'נוכחית' כך שהוא יצביע X בתים קדימה.
            3. שים בגודל של החוליה הנוכחית את X.
        iii. גורמים לראש להצביע על החוליה הקודמת.

iv. מחזירים למשתמש את הכתובת שנמצאת מיד לאחר ה-Header של החוליה הנוכחית.

2. אם החוליה הנוכחית היא הראש, סימן שעברנו על כל הרשימה ויש לבקש בלוק זיכרון נוסף ממקצה הזיכרון הגדול. קוראים לפונקציית "הגדל זיכרון" שמוסיפה חולייה חדשה מכילה בלוק זיכרון גדול לרשימה, ומחזירה חולייה כך שהחולייה->הבא מצביע לחולייה החדשה.

אם הפעולה נכשלה, מחזירים null. אחרת גורמים לחוליה הנוכחית להצביע לחוליה שהפונקציה החזירה.

3. גורמים לחוליה הקודמת להצביע לחוליה הנוכחית ולחוליה הנוכחית להצביע לחוליה שאחרי החוליה הנוכחית. ממשיכים את הלולאה האינסופית.

פונקציית הגדלת הזיכרון (פונקציה פנימית) שמקבלת את X ומחזירה חולייה כך שחולייה->הבא היא החוליה שמכילה בלוק זיכרון גדול חדש:

1. מעגלים את X למעלה לכפולה של בלוק מינימלי (בפועל גודל של דף זיכרון – 4KB) – נסמן גודל זה ב-Y.
2. מבקשים Y בתים מהמקצה זיכרון הגדול. אם הפעולה נכשלה, מחזירים null. אחרת יוצרים מצביע חוליה שמצביעה לבלוק הזיכרון שהתקבל.
3. מאתחלים את גודל החוליה ל-Y.
4. מכניסים את החוליה החדשה לתחילת הרשימה:
   1. גורמים לחוליה->הבא להצביע על הראש->הבא.
   2. גורמים להראש->הבא להצביע על החוליה החדשה.
5. מחזירים את הראש.

פונקציית שחרור זיכרון – מקבלת מצביע לכתובת שהוקצתה בעבר וגורמת לאזור הזיכרון להיות פנוי (כלומר, קריאה להקצאת זיכרון פעם הבאה יכולה להחזיר כתובת באזור זה). מטרת הפונקציה היא להכניס את החוליה ששייכת לזיכרון שהוקצה לרשימת החוליות הפנויות, תוך מיזוג עם חוליות קיימות על מנת למנוע יצירת חורי זיכרון.

1. מחסרים מהכתובת זיכרון את הגודל של Header ויוצרים חוליה שיצביע לכתובת שהתקבלה. נקרא לחולייה זו הבסיס.
2. יוצרים חולייה נוכחית שמצביע לתחילת הרשימה (הראש).
3. מוצאים את 2 החוליות שהבסיס נמצא ביניהם – מקדמים את החולייה הנוכחית כל עוד כתובת של הבסיס קטנה מהכתובת החוליה או גדולה מהכתובת של החולייה->הבא. אם הכתובת של החולייה->הבא קטן מכתובת החולייה (מצב שקורה כאשר מגיעים חזרה לתחילת הרשימה, או מגיעים לתחילה של בלוק זיכרון גדול נפרד), יוצאים מהלולאה אם כתובת הבסיס היא אחרי כתובת החולייה->הבא ולפני כתובת החולייה.

4. ממזגים את הבסיס עם החוליות שהוא נמצא ביניהם (חיבור "חורי" זיכרון):
   1. אם הבסיס מיד לפני החוליה אחריו, ממזגים את החוליה אחריו לתוכו - מגדילים את גודל הבסיס בגודל החוליה שאחריו וגורמים לבסיס->הבא להצביע לחוליה אחריו->הבא. אחרת גורמים לבסיס->הבא להצביע לחולייה אחריו.
   2. אם הבסיס מיד אחרי החוליה לפניו, גורמים לחולייה לפניו להכיל את הבסיס - מגדילים את גודל החוליה לפניו בגודל הבסיס ומגורמים לחולייה לפניו->הבא להצביע לבסיס->הבא. אחרת גורמים לפני הבסיס להצביע לבסיס.

5. גורמים לראש להצביע לחולייה שלפני הבסיס.

סיבוכיות הן ההקצאה והן השחרור של הזיכרון הוא O(n) במקרה הגרוע. ההקצאה היא מסוג של First-Fit כך שהיא מחזירה את בלוק הזיכרון הראשון שעונה לדרישות של המשתמש, ולכן במקרה הכללי ההקצאה לא תיקח זמן לינארי (אין חיפוש של הבלוק ה"טוב" ביותר, אלא הראשון שמתאים).

יתרונות האלגוריתם:

1. קריאה להקצאת הזיכרון מפצלת בלוקים גדולים של זיכרון לבלוקים קטנים סמוכים ולכן מונעת קריאות מיותרות להקצאה של זיכרון גדול חדש.
2. קריאה להקצאת זיכרון מבצעת לולאה רק על חוליות פנויות, ולכן נמנע חיפוש מיותר באזורים שכבר הוקצו. בנוסף, בלוקים גדולים יותר לרוב יאכלסו את תחילת הרשימה, לכן החיפוש בדרך כלל יצליח בשלבים מוקדמים.
3. קריאה לשחרור הזיכרון מסוגלת לשלב חזרה בלוקים של זיכרון שהיו סמוכים, וכך מונעת יצירת חורים מיותרים בזיכרון.
4. קריאה לשחרור זיכרון גורמת לאזור הזיכרון ששוחרר להתווסף לתחילת הרשימה, והיות שאזור זה מתמזג חזרה עם בלוק שבעבר היה גדול, הקריאה מגדילה הסיכויים להקצאה מוצלחת בפעם הבאה.

אלגוריתמים נוספים לפתרון הבעיה:

1. שימוש ב-bitmap שמציין אם בלוק זיכרון בגודל קטן יחסית וקבוע פנוי. החיסרון בגישה הזו היא שה-bitmap לוקח יותר מקום ככל שהוא ממפה בלוקים קטנים יותר. המטרה היא למפות בלוקים בגודל של בייטים בודדים, ולכן bitmap כזה יקח המון מקום (לא פרקטי). בנוסף, חיפוש בלוק פנוי יקח המון זמן (כי מדובר בלולאה על כל הזיכרון הפנוי).

2. שימוש ברשימות מקושרות ממוינות – אלגוריתם דומה לאלגוריתם המוצג - יש חוליות שמציינות בלוקים פנויים, והרשימה ממוינת לפי גודל הבלוקים. כאשר מבקשים זיכרון פנוי, מחזירים את החוליה הקטנה ביותר שעונה על הדרישות (כלומר, מדובר באלגוריתם Best-Fit). בצורה כזאת לא מבזבזים זיכרון מיותר. כאשר משחררים זיכרון, מוסיפים את החוליה לרשימה וממיינים אותה מחדש. החיסרון בשיטה זו היא שהיא דורשת מיון רשימה מקושרת בכל הקצאה והוצאה, אבל היתרון בה היא שמספר חורי הזיכרון הוא מינימאלי בכל שלב.

ממשק משתמש

בניית מערכת ההפעלה

בניית מערכת ההפעלה יכולה להיעשות רק בלינוקס. תהליך הבנייה דורש קרוס קומפיילר של GCC 5.1.0 ו-QMake (מערכת בנייה שמגיעה עם Qt Creator) ותהליך העתקת המערכת להתקן חיצוני (כגון DOK) דורש מספר כלים נוספים. על מנת להקל על תהליך התקנת הכלים הנדרשים, הפרויקט מספק סקריפט בשם install-build-tools.sh המיועד לעבוד על הפצות לינוקס מבוססות Debian.

בתור משתמש שרוצה לבנות את המערכת, תהליך בניית המערכת יהיה:

```
$ cd scripts
$ ./install-build-tools.sh
```

לאחר מכן, נכנסים ל-Qt Creator (שאמור להיות מותקן אם הוא עוד לא היה מותקן קודם) ופותחים את קובץ הפרוייקט Bug.pro. לאחר מכן לוחצים על כפתור הבנייה. בשלב הזה מערכת ההפעלה קומפלה לקובץ תמונה בינארי שניתן יהיה לכתוב להתקן חיצוני.

התקנה על התקן חיצוני (DOK)

על מנת להתקין את המערכת על DOK שניתן יהיה להריץ בצורה חייה מהמחשב, מחברים את ה-DOK למחשב ומריצים:

```
$ cd scripts
$ ./create-usb.sh
```

כדאי לשים שהסקריפט מניח שה-DOK נמצא ב-/dev/sdb. יש להחליף בסקריפט את הנתיב עם נתיב אחר במידה וה-DOK לא נמצא ב-/dev/sdb. כמו כן, חשוב לדעת שההתקנה תמחק את כל מה שהיה ב-DOK.

בשלב הזה ה-DOK מכיל את כל מערכת ההפעלה והוא bootable.

תפעול המערכת

על מנת להריץ את המערכת, יש לכבות את המחשב ולחבר את ה-DOK. בעליית המחשב, יש לבחור ב-BIOS את האופציה להעלות את המחשב מה-DOK.

לאחר שהמערכת עלתה, המשתמש יראה מולו את ה-shell הבא:

```
                    Welcome to Bug OS!
          * You can switch shells by pressing F1-F12 *

Bug shell started (pid 3)
Please enter your name: avidan

Hello, avidan!
Type 'help' to view list of commands
[/avidan/]: help
list      - list files in current directory
cd        - change current directory
view file - view content of file
type file - type to file
tasks     - list all tasks running
kill pid  - kill task with specified pid
clear     - clear console
date      - view current date
sleep ms  - sleep for 'ms' millisecs
mem       - view physical memory statistics
exit      - exit current shell
snake :)  - play snake
[/avidan/]:
```

עם עליית המערכת, רצים במקביל 12 shellים שניתן להחליף ביניהם בכל שלב ע"י לחיצה על F1-F12. עובדה זו מאפשרת למשתמש להריץ תוכנות רבות במקביל. כל shell שמתחיל לרוץ שואל את שם המשתמש, ונכנס לתיקייה שמיועדת עבור המשתמש. המטרה בכך היא לאפשר למספר משתמשים שונים להשתמש במחשב, שלכל אחד מרחב שמיועד עבורו.

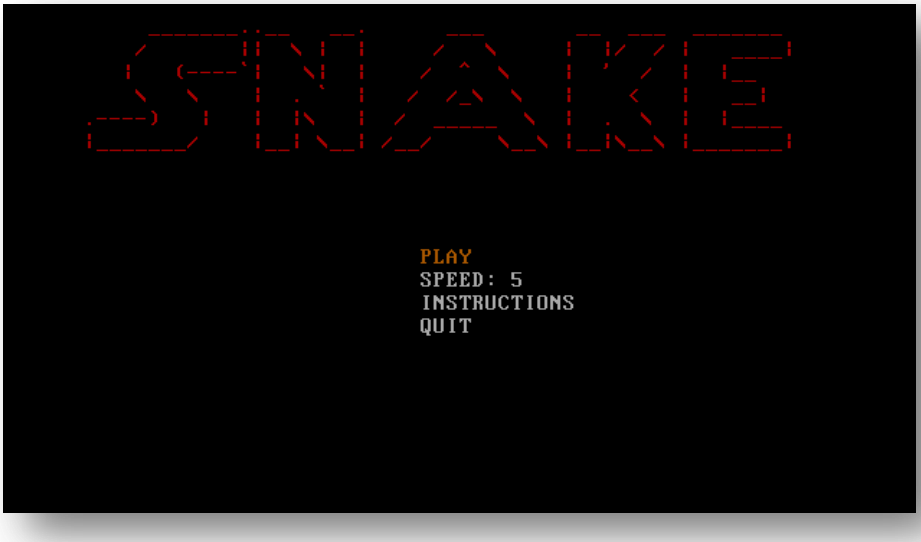
על מנת לקבל מידע על הפקודות השונות שאפשר להריץ ב-shell, ניתן לכתוב help בכל שלב.

מעט רקע על ניהול תהליכים – המשתמש יכול לראות את התהליכים שרצים כרגע באמצעות הפקודה tasks, והוא גם יכול להרוג כל תהליך אחר, רק בתנאי שזה לא תהליך ששייך ל-kernel. למשל, במקרה הבא:

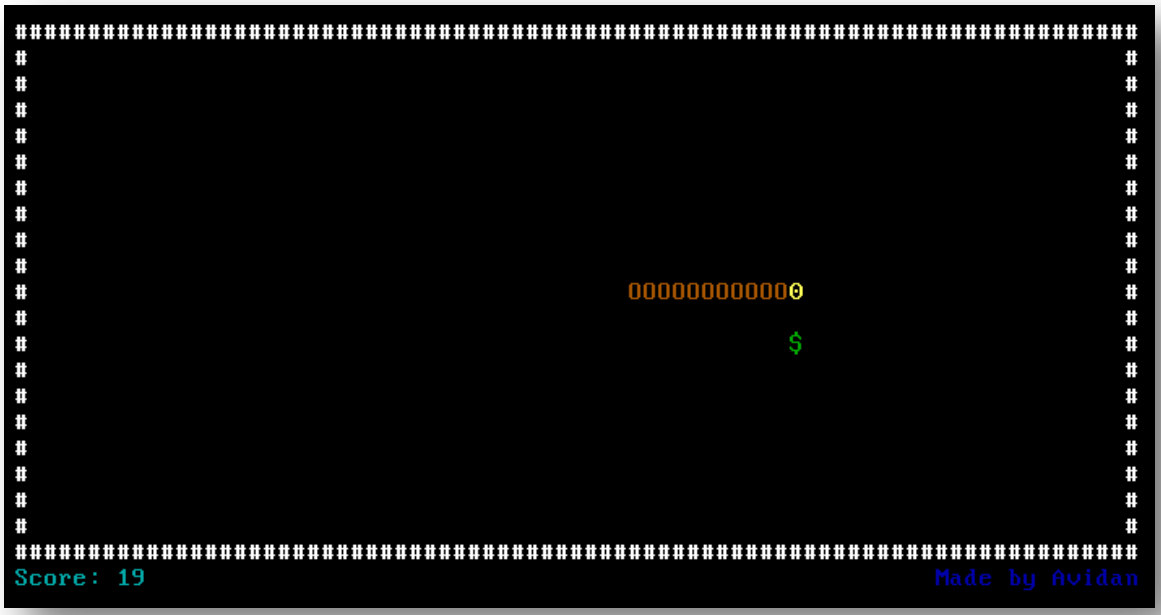
```
[/avidan/]: tasks
idle: kernel pid=0 parent=-1 tty=0 state=READY wd=/
cleaner: kernel pid=1 parent=-1 tty=0 state=BLOCKED wd=/
init: kernel pid=2 parent=-1 tty=0 state=WAITING wd=/
shell: user pid=3 parent=2 tty=0 state=RUNNING wd=/avidan/
shell: user pid=4 parent=2 tty=1 state=BLOCKED wd=/
shell: user pid=5 parent=2 tty=2 state=BLOCKED wd=/
shell: user pid=6 parent=2 tty=3 state=BLOCKED wd=/
shell: user pid=7 parent=2 tty=4 state=BLOCKED wd=/
shell: user pid=8 parent=2 tty=5 state=SLEEPING wd=/john/
shell: user pid=9 parent=2 tty=6 state=WAITING wd=/victor/
shell: user pid=10 parent=2 tty=7 state=BLOCKED wd=/danny/work/
shell: user pid=11 parent=2 tty=8 state=BLOCKED wd=/
shell: user pid=12 parent=2 tty=9 state=WAITING wd=/bobby/
shell: user pid=13 parent=2 tty=10 state=BLOCKED wd=/
shell: user pid=14 parent=2 tty=11 state=BLOCKED wd=/
snake: user pid=15 parent=12 tty=9 state=BLOCKED wd=/bobby/
snake: user pid=16 parent=9 tty=6 state=BLOCKED wd=/victor/
[/avidan/]:
```

ניתן לראות שהמשתמש יכול להרוג את כל התהליכים מ-3 עד 16 כולל, אך לא את התהליכים 0-2 שרצים ע"י ה-kernel. בנוסף, ניתן לראות מה התיקייה הנוכחית של כל תהליך, מצב התהליך, התהליך האבא, ומספר הטרמינל שבו התהליך כרגע רץ.

הוראות לשימוש בסנייק: לאחר שכותבים בקונסול את הפקודה snake, נפתח החלון הבא:



על מנת לנווט בתפריט (לשחק, לשלוט על המהירות, לקרוא את ההוראות או לצאת) ולשחק, יש להשתמש במקשי החצים WASD.

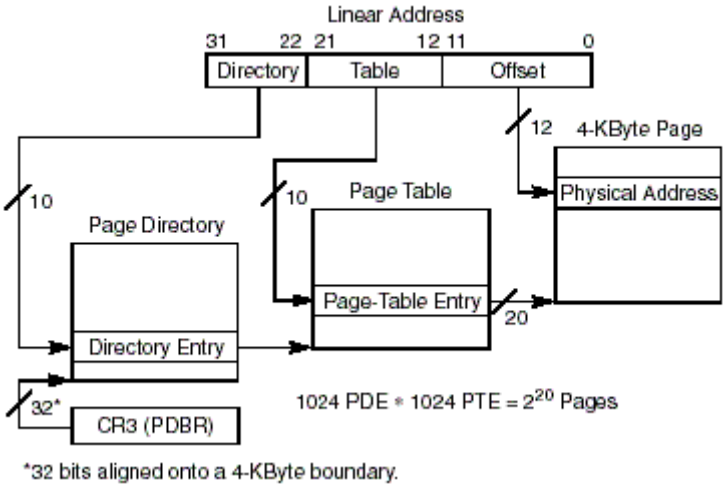


הוראות מלאות על המשחק ניתן למצוא ב-Instructions, אך בגדול המטרה היא שהנחש לא יפגע בעצמו ובקירות, ועליו לאכול אוצרות (פלוסים ודולרים) על מנת לקבל עוד נקודות.

רפלקציה אישית

כל פיתוח המערכת היה מלווה בלמידה של נושאים חדשים, שאת רובם לא הכרתי. פיתוח המערכת גרם לי למעשה להבין את כמות העבודה העצומה המושקעת למימוש כל חלק קטן במערכת הפעלה שנראה טריוויאלי לנו בתור משתמשים וגם בתור מתכנתים. בסביבה שבה לא ממומש שום דבר, אני הייתי אחראי למימוש של הכל – וללא ספק למדתי מכך המון.

בין הנושאים הכי מאתגרים הכרוכים בפיתוח מערכת הפעלה ניתן למצוא את ה-Paging וניהול הזיכרון - נושא שקשור כמעט לכל חלק במערכת, ומסובך מאוד לדיבוג. טכנולוגיית ה-Paging היא טכנולוגיה שבאה עם ה-32Bit Protected Mode, ומאפשרת למעשה קיום של "זיכרון וירטואלי" – כל תהליך שרץ חושב שכל מרחב הזיכרון שייך לו (למעט המרחב ששמור ל-kernel), ואין לו דרך לגשת לזיכרון של תהליכים אחרים. מאחורי הקלעים, בכל גישה לזיכרון מתבצע מעבר מכתובת וירטואלי לכתובת זיכרון פיזית באמצעות טבלה מיוחדת:



הכתובת הפיזית של הטבלה שמורה באוגר מיוחד ששמו CR3. הטבלה עצמה מורכבת מ-1024 "רשומות תיקייה" שיכולות להצביע על כתובת פיזית של טבלת דף כלשהי. כל טבלת דף מכילה 1024 "רשומות דף" שיכולות להצביע על הכתובת הפיזית של דף הזיכרון (כל דף זיכרון הוא בגודל 4KB). בצורה הזו כל כתובת זיכרון וירטואלית (בגודל 32-Bit) עוברת תרגום ע"י פיצול הכתובת לאינדקסים המתאימים של רשומת התיקייה ורשומת הדף, ובמידה והטבלה מכילה את המיפוי הרצוי, מוחזרת הכתובת הפיזית של דף הזיכרון ואליו מתווסף ההיסט של הכתובת מתחילת של דף.

לאחר שפיתחתי חלק גדול מהמנגנון, נתקלתי בבעיה לא פשוטה – לאחר שמצליחים למפות את ה-kernel לעצמו ו"מפעילים" את מנגנון ה-Paging – כיצד ניתן לערוך לאחר מכן את הנתונים בטבלה? הרי כעת הזיכרון במצב וירטואלי, ולא ניתן לגשת דרך כתובות פיזיות לרשומות התיקייה או רשומות הטבלה (והמצביעים היחידים שקיימים בטבלה הם מצביעים פיזיים ולא וירטואליים).

פתרון מסוים, שניתן למצוא במדריכים רבים לפיתוח מערכות הפעלה, הוא לשמור ערכים כפולים – לאחר טבלת רשומות התיקייה הפיזית לשמור כל הזמן טבלה של רשומות תיקייה וירטואליות שימופו לכתובות הפיזיות המקבילות להן וגם לשמור את הכתובת הווירטואלית של הטבלה עצמה. החיסרון

הפתרון הזה הוא שהוא דורש הקצאת זיכרון נוספת לכל מיפוי שנעשה, והוא גם מאוד מסובך (במיוחד כאשר מגיע החלק שצריך לממש תהליכים שמקבלים טבלאות נפרדות).

הדרך שבה אני פתרתי את הבעיה היא באמצעות פתרון אלגנטי במיוחד (לדעתי). מכיוון שהפורמט של רשומות תיקייה ורשומות דף הוא פורמט כמעט זהה, ניתן למעשה "לגרום" למעבד לחשוב שמה שבפועל הוא רשומת תיקייה, להיות רשומת טבלה. הדרך שבה זה עובד היא ברשומת התיקייה האחרונה (1023) שמים את הכתובת של טבלת התיקיות עצמה. בצורה כזו ניתן לגשת ישירות הן לרשומות התיקייה והן לרשומות הטבלה ע"י שימוש בזיכרון וירטואלי בלבד:

1. על מנת לגשת לטבלת התיקיות, ניגשים לכתובת האחרונה בזיכרון. המעבד מפרק את הכתובת לאינדקסים של התיקייה והדף, ששניהם יהיו 1023. הוא ייגש לרשומה מספר 1023 בטבלת התיקיות, ויראה את הכתובת שהיא מצביע אליו ויחשוב שכתובת זו היא מייצגת טבלת דפים. הוא ייגש לכתובת זו (שבפועל היא התיקייה עצמה) ויגש לרשומת התיקייה מספר 1023, שזאת שוב תצביע לכתובת הפיזית של התיקייה עצמה!

2. על מנת לגשת לטבלת הדפים, ניגשים לכל כתובת שרשומת התיקייה שלי היא 1023, למשל על מנת לגשת לטבלה מספר 5, ניגשים לכתובת שרשומת התיקייה שלה היא 1023 ורשומת הדף שלה היא 5 (0xffc0500). המעבד ניגש לרשומת התיקייה מספר 1023, ורואה את הכתובת שהוא מצביע אליה ויחשוב שכתובת זו מייצגת טבלת דפים. הוא ייגש לכתובת זו (שבפועל היא התיקייה עצמה) וייגש לרשומת התיקייה מספר 5 שתצביע לכתובת הפיזית של טבלת הדפים המתאימה.

אני אישית מאוד התלהבתי מהפתרון הזה, כיוון שהסתבכתי במשך הרבה זמן עם מימוש טוב של API ל-Paging. הפתרון הוא כל כך אלגנטי שהוא עבד בצורה חלקה גם כאשר התחלתי לממש את ה-Scheduler ונדרשתי ליצור לכל תהליך טבלת מיפוי משלו – כל תהליך ממפה את התיקייה שלו לעצמו באותה דרך, ואין שום צורך לשמור מידע נוסף על כל תהליך.

המוצר המוגמר אל מול התכנון הראשוני

הפיצ'ר	תכנון ראשוני	מוצר מוגמר
Bootloader ייעודי	✓	✓
זיהוי מפת הזיכרון הפיזי	✓	✓
כתיבה למסך	✓	✓
פסיקות תוכנה	✓	✓
פסיקות חומרה	✓	✓
Segmentation	✓	✓
Paging	✓	✓
מקצה זיכרון פיזי	✓	✓
מקצה זיכרון וירטואלי	✓	✓
Heap	✓	✓
Scheduler וריבוי תהליכים	✓	✓
User Mode	✓	✓
קריאות מערכת	✓	✓
דרייבר מקלדת ו-IO	✓	✓
טרמינלים מרובים	⊗	✓
דרייבר שעון	⊗	✓
מערכת קבצים וירטואלית	✓	✓
מערכת קבצים פיזית (דרייבר FAT12)	✓	⊗
ספרייה סטנדרטית	⊗	✓
ספריית User Space	⊗	✓
טעינת תהליכי User מ-ELF	✓	✓
Shell	✓	✓
Snake	⊗	✓
GUI	✓	⊗
דרייבר לכרטיס רשת	✓	⊗

2 הפיצ'רים האחרונים היו בתכנון הראשוני בעיקר ברמת התיאוריה – לא ביצעתי אותם מכיוון שהמימוש שלהם דורש יותר מידי עבודה.

את מערכת הקבצים הפיזית, שבתכנון הראשוני הייתה אמורה להיות דרייבר להתקן חיצוני (שיכול להיות ה-DOK ממנו עלתה המערכת) עם מערכת קבצים FAT12, לא מימשתי מאותה סיבה. מדובר בהרבה עבודה ולא היה לי מספיק זמן. הקבצים במערכת ההפעלה לא נשמרים כרגע בהתקן פיזי.

ביבליוגרפיה

המדריך הבסיסי שעקבתי אחריו ללמידה על מימוש הקונספטים הבסיסיים במערכת הפעלה:

http://jamesmolloy.co.uk/tutorial_html/

המקור העיקרי והמקיף ביותר ללמידה של כל מה שצריך לפיתוח מערכת הפעלה:

http://wiki.osdev.org/Main_Page

הפורומים של ה-wiki:

http://forum.osdev.org/

מדריך מקיף לכתיבת Bootloader שקורא מ-FAT12:

http://www.independent-software.com/writing-your-own-toy-operating-system/

מקורות נוספים:

http://www.julienlecomte.net/simplix/

http://www.freebsd.org/doc/en/books/arch-handbook/book.html#boot

https://github.com/guilleiguaran/xv6

http://www.brokenthorn.com/Resources/OSDevIndex.html

http://board.flatassembler.net/topic.php?t=12389