Technologic Papers

היכרות עם קבצי ריצה - חלק ראשון

מאת עידו קנר

הקדמה

בסדרת מאמרים זו אנסה להסביר על סוגי הריצה השונים של קבצי הרצה, הבדלים בין צורות ופורמטים שונים של קבצים אלו, ההבדלים בין ריצת Byte Code לבין Native Execution ואף להבין מעט יותר כיצד הקרנל ואף המעבד מתמודדים עם הנושא.

הכותב מניח כי לקרוא יש הבנה כלשהי בעולם התכנות, ובמערכות הפעלה, אך אינו דורש הבנה עמוקה בנושא זה.

בין ריצה להרצה

כפי שלומדים רבים בתחום מדעי המחשבים, המחשב נועד על מנת לבצע פקודות, או סדרת פעולות. במקור היו אלו פעולות חישוביות בלבד, וכיום הנושא מורכב יותר. כאשר המחשב מקבל הוראות ריצה לביצוע, מאחורי הקלעים, המחשב אינו יודע מה זה תכנות מונחה עצמים, מה זו שפה דינאמית, וכיצד פונקציות מונדיות עובדות.

טכנולוגיות אלו, קשורות לשפות תכנות, אשר מסייעות למתכנתים ליצור מערכות מורכבות, בצורות מופשטות יותר, ויש איזשהו "כלי" אשר מתרגם אותם לתוצר שיכול לרוץ. אך ההרצה מתבצעת על ידי משהו אחר, ואותו "משהו" משתנה לפי סוג השפה או המימוש של השפה, אשר נעשה בה שימוש.

לפני ההסבר על הסוגים השונים של הריצה, אסביר בקצרה מה קורה למחשב בפועל כאשר יש ריצה של הקוד שנכתב.

כיצד המעבד עובד

המחשב, או יותר נכון המעבד, יודע לבצע פעולות על זיכרון, וכל מעבד מכיל סט פעולות שנתמכות בו, והן נקראות Instruction Set Architecture או ISA בקיצור.

עבור המעבד, כל דבר הוא כתובת זיכרון. זה אומר כי מסך, כרטיס אודיו, דיסק, עכבר, וכל חומרה אחרת, כולם בעצם כתובות זיכרון. תפקיד מערכת הפעלה, הוא לקחת את כתובות הזיכרון ולעשות איתן דברים. למשל, להבין מה המידע שיש על דיסק ולהתאים מערכת קבצים, לאותו המידע, אם בכלל הוא קיים.

על מנת לעשות זאת באופן יעיל, מרבית מערכות ההפעלה המקובלות בשוק מפרקות אזורי זיכרון שונים לחלקים שונים. כל פעולה על הזיכרון אשר נתמכת על ידי המעבד, מקבלת ערך בינארי כלשהו אשר המעבד מבין, ופקודה מקבילה של שפה בשם Assembler אשר מאפשרת לבני אדם להבין אותה.

לדוגמא, להדפיס Hello World בלינוקס עבור מעבד x86_64 יראה כך:

```
section     .text
global      _start  ;must be declared for linker (ld)

_start:  ;tell linker entry point

    mov     edx,len    ;message length
    mov     ecx,msg    ;message to write
    mov     ebx,1      ;file descriptor (stdout)
    mov     eax,4      ;system call number (sys_write)
    int     0x80       ;call kernel interrupt

    mov     eax,1      ;system call number (sys_exit)
    mov     ebx,0      ;exit with error code 0
    int     0x80       ;call kernel interrupt

section     .data

msg     db  'Hello, world!',0xa ;our dear string
len     equ $ - msg             ;length of our dear string
```

[מבוסס על קוד מאתר: http://asm.sourceforge.net/intro/hello.html]

אך כאשר מדובר בקוד Hello World בלינוקס למעבד ARM הוא יראה בכלל כך (התחביר הוא ב-GAS):

```
.data
msg:
    .ascii      "Hello, World!\n"
len = . - msg

.text

.globl _start
_start:
    /* syscall write(int fd, const void *buf, size_t count) */
    mov     %r0, $1     /* fd -> stdout */
    ldr     %r1, =msg   /* buf -> msg */
    ldr     %r2, =len   /* count -> len(msg) */
    mov     %r7, $4     /* write is syscall #4 */
    swi     $0          /* invoke syscall */

    /* syscall exit(int status) */
    mov     %r0, $0     /* status -> 0 */
    mov     %r7, $1     /* exit is syscall #1 */
    swi     $0          /* invoke syscall */
```

[מבוסס על קוד מאתר: http://peterdn.com/post/e28098Hello-World!e28099-in-ARM-assembly.aspx]

תחביר GAS שבתמונה, שינה מעט מאוד, אבל הוראות המעבד הן לגמרי שונות.

השוני בין הקריאות שונה בגישה. למשל ב-x86_64, קריאות לפעולות של syscalls מתבצעות באמצעות פסיקות (פעולת int או interrupt בשם המלא), שהן דרך להודיע על "אירוע" שצריך להתרחש. בעוד שהקריאה ב-ARM מבצעת Exception תוכנתי בשביל לבצע קריאה לפעולת Syscall, על ידי שימוש בפקודה swi.

השימוש של שניהם למעשה, עושה אותו הדבר, יציאה מהמיקום הנוכחי של הזיכרון, וקריאה לקוד חיצוני שתופס מיקום זיכרון אחר. אפשר גם לראות כי השימוש של שניהם הוא להזין ערכים שונים לאוגרים.

כאשר מדובר בשפה גבוהה יותר, אשר אינה תלויה במעבד, כדוגמת C, יש כבר הפרדה בדרך כלל בין דברים. כלומר למתכנת אין "הרגשה" של שינוי כתובות זיכרון, ואין צורך לדעת כיצד המעבד ידע להריץ Syscalls, ובמקום זאת ההתמקדות היא פשוטה יותר עבור מגוון ענק של מעבדים:

```c
#include <stdio.h>

int main() {
  printf("Hello World!\n");
  return 0;
}
```

החיסרון העיקרי של כתיבה ב-C, היא שקוד הכתוב ב-C, למרות שנראה קצר יותר, מכיל בתוכו יותר "זבל", אשר נכנס לריצה, ובכך מגדיל מאוד את קובץ הריצה.

כאשר מערכת ההפעלה רוצה להריץ את הקוד, היא לרוב תטען את כולו לזיכרון ורק אז תנסה לעקוב אחרי ההוראות שיש. על מנת שמערכת ההפעלה תצליח לעשות זאת, היא צריכה לתמוך במספר סוגים של תצורות ריצה. לשם כך, נוצרו קבצי ריצה שונים, אשר לרוב מבצעות את אותה הפעולה, רק בדרכים שונות.

מפרש, מהדר ומקשר

בפרק הקודם הצגתי קוד המבצע הדפסת הודעה בלינוקס. בשביל שהקוד ירוץ, הוא חייב לעבור מספר פעולות:

1. מעבר ופירוש של הקוד (Code Parsing)
2. תרגום בינארי של פקודות (Compiling)
3. קישור בינארי לתצורת ריצה (Linking)

מה היא תצורת ריצה?

יש הוראות מעבד, אשר לרוב מותאמות למערכת הפעלה מסוימת. הוראות אלו צריכות להישמר בצורה שמערכת ההפעלה תדע להפעיל אותן, לאתחל עבורן זיכרון, ובמידה ויש תלויות שונות, לדעת עליהן ולדעת להתאים אותן עם כתובת דינמית לריצה הנוכחית של המערכת.

לשם כך המציאו סוגי קבצים שונים, והם בעצם תצורות הריצה עליהן אני מדבר.

ישנם הרבה פורמטים של תצורות ריצה. רובם מבצעים אותו הדבר, בצורה שמערכת ההפעלה יכולה לנהל, ויש פורמטים אשר קרובים יותר לתצורה בה המעבד עובד, ולא תמיד זקוקים להרבה פעולות מצד מערכת ההפעלה בשביל לרוץ.

אך לפני שאדבר על מה הן תצורות השונות (לפחות על חלקן), חשוב להבין יותר על התהליך שמתבצע: חושב לי לציין, כי ישנן שלוש סוגי שפות תכנות עיקריות כאשר מדובר בריצה, ישנן שפות **מקומפלות** (כדוגמת C), ישנן שפות אשר **מפורשות** בזמן ריצה (כדוגמת פיתון Javal), וישנן שפות שהן בעצם **הוראות** לתוכנה (כדוגמת SQL).

כל סוג שפה שכזו, מכילה מסלול מעט שונה, ובחלק הזה של המאמר, אדבר רק על שפות מקומפלות, אך בחלק אחר, ארחיב גם על סוגי השפות הנוספות.

מפרש

כלל שפות התכנות אשר אני מכיר, מכילות מערכת אשר יודעת לפרש את הכתוב לצורה קלה יותר להבנה. מכאן השם "מפרש" או Parser באנגלית. הפעולה עצמה קיבלה את השם Lexical Parsing.

הפעולה שהמפרש עושה היא לקרוא תווים, ולהתחיל להפוך אותם לאסימונים (tokenization) שונים. הניתוח עצמו מזהה כל תו או קבוצה של תווים לסוג שלהם, בהתאם למבנה מסוים, כולל היכולת לדעת למשל האם זה חלק בלוק ריצה, או האם זה עומד בפני עצמו. למשל ביטויים מתמטיים פשוטים, כדוגמת הביטוי הבא:

```
a = a + 5
```

יתורגם בגישה הבאה:

```
(var a
  (operator =
    (var a)
    (operator +)
    (const 5)
  )
)
```

התרגום הזה, יוצר עץ מסוים אשר מאפשר להבין כי יש משתנה, יש אופרטור, וכיוב', ומה שייך לאיזו פעולה. בזמן הניתוח, מגיע גם שלב דיווח שגיאות במידה ויש. למשל: "משתנה בשם a לא הוגדר אבל אתה מנסה לגשת אליו".

שפות כדוגמת C, זקוקות לפעולה נוספת של ניתוח אסימונים, על מנת להוסיף קבצי include, תרגום של מאקרו, וכו'. פעולה זו נקראת Pre-Processor. שפות כדוגמת C++ צריכות בנוסף ל-Pre-Processor, פירוש מרובה (נקרא multi-pass או wide pass) על מנת לנתח שכל המידע הדרוש מתקיים. כל ניתוח מתבסס

על הניתוח הקודם על מנת ליצור תמונה שלמה של התחביר, עד למעבר האחרון. במידה ולאחר המעבר האחרון, הכל תקין יש ניתוח של המפרש לאסימונים מלא, ובמידה ויש בעיה, יש על כך דיווח.

מרבית שפות התכנות בשוק אינן דורשות יותר ממעבר בודד, אך בכל זאת כאמור, ניתן למצוא שפות מורכבות יותר.

מהדר

לאחר שיש מושג תוכנתי מה כתוב בקוד המקור, יש כלי נוסף אשר נמצא בשימוש, והוא נקרא מהדר (compiler). במקרה של מאמר זה, תפקיד המהדר, הוא למעשה להמיר לשפת אסמבלי את התוצר של התוכנה.

אחזור רגע להדגמה בשפת C של הצגת המחרוזת:

```
#include <stdio.h>

int main() {
  printf("Hello World!\n");
  return 0;
}
```

המהדר יתרגם את זה לקוד הבא:

```
     .file    "hello.c"
    .section .rodata
.LC0:
    .string    "Hello World!"
    .text
    .globl   main
    .type    main, @function
main:
.LFB0:
    .cfi_startproc
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    leaq    .LC0(%rip), %rdi
    call    puts@PLT
    movl    $0, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size    main, .-main
    .ident    "GCC: (GNU) 7.2.0"
    .section    .note.GNU-stack,"",@progbits
```

את התוצר השגתי באמצעות הרצה של gcc במקרה הזה, בצורה הבאה:

```
$ gcc -Wall -s hello.c
```

ההרצה יוצרת קובץ בשם hello.s אשר מכיל את התרגום הישיר לקוד האסמבלי (בתחביר GAS).

במידה ויש הכרות עם שפת אסמבלי למעבדי אינטל, ניתן לראות כי התחביר הוא עבור x86_64, בשל חלק מהאוגרים במקרה הזה, הנמצאים בפועלה. התרגום הזה לשפת מכונה, מאפשר עכשיו להתמקד בביצוע עצמו, כלומר מה מערכת ההפעלה והמעבד בעצם צריכים לבצע.

בעולם היוניקס, ישנם שני תחבירים עיקריים אשר המהדר יכול ליצור עבורם את קוד האסמבלי. הראשון הוא תחביר Intel והשני הוא תחביר GAS. התחביר השני הוא עבור מהדר בשם GNU as, ולכן קיבל את הקיצור GAS.

מהדר as וכן מהדר Intel עבור אסמבלי, הם מהדרים אשר ממירים את קוד האסמבלי שנוצר מהמהדר הראשון, לשפת מכונה. אך, gcc אינו חייב ליצור קובץ אסמבלי אשר ירוץ באמצעות as או מהדר אסמבלי אחר. למעשה בדרך כלל, gcc יצור קובץ אובייקט (object file), אשר מכיל שפת מכונה עם הפקודות הנדרשות, והוא יהיה מוכן בפורמט בינארי כלשהו על מנת ליצור ממנו קובץ תצורת ריצה.

מקשר

לאחר פירוש שפת התכנות למידע בינרי, מגיע החלק של כלי בשם Linker, או מקשר בעברית. התפקיד של המקשר, הוא לקחת את המבנה הבינרי שנוצר על ידי הקומפיילר, ולהמיר אותו לתוכן שרוצים, שהוא לרוב קבצי ריצה או ספריות דינמיות (תצורת ריצה). בשביל להפיק את התוצר, המקשר לוקח את כל קבצי האובייקטים השונים, ומצרף אותם לפי הוראות מסוימות לקובץ אחד אשר צריך אותם בשביל התוצר הסופי.

המקשר יכול ליצור תצורת ריצה מסוימת (עליהן אתחיל להסביר בפרק הבא), או קוד אשר פשוט מתורגם לקריאות מערכת, ואינו תלוי במערכת ההפעלה. הסיבה לכך שיש קוד אשר אינו תלוי במערכת הפעלה, היא היות וכאשר רוצים ליצור קוד שהוא מערכת הפעלה, או ריצה ישירה מול המעבד (כאשר מדובר במערכת embedded פשוטה, אשר אינה דורשת מערכת הפעלה), יש צורך ליצור קוד אשר ידע לרוץ ישירות מול המעבד, והיכולת הזו מאפשרת לספק זאת.

כאשר מדובר בקישור לתצורות ריצה של מערכות הפעלה, ישנן שתי צורות קישור עיקריות:

- קישור סטטי
- קישור דינמי

קישור סטטי מאפשר לקחת כל קוד הנמצא בשימוש ולהכניס אותו לקובץ ריצה בודד. במידה והמקשר מתוחכם מספיק (לרוב בסיוע המידע שהתקבל מהקומפיילר), הוא ידע להכניס רק את מה שנמצא בפועל בשימוש, ורק התלויות האלו, יכנסו לקובץ הריצה. במידה והמקשר פחות מתוחכם, הוא יכניס ספריות

שלמות לתוך קובץ הריצה. לפעמים גם יש מצבים בהם פונקציה שהקוד שלנו עושה בה שימוש, זקוק לתלויות נוספות, ואז גם הם יכנסו לקובץ הריצה.

היתרון הוא, שניתן לספק קובץ אחד שירוץ ואין צורך לדאוג אם הסביבה שבה הקובץ רץ מספקת את התלויות השונות או לא. יותר מזה, היא אינה תלויה גם בגרסה שיש בסביבה בה רץ הקובץ. למשל גרסה חדשה או ישנה יותר ממה שהקובץ עצמו צריך, היות והכל נמצא בתוכו.

אך יש בעניין זה גם מספר חסרונות:

- ראשית, הקובץ מאוד גדול, יחסית, והוא כולו נטען בזיכרון. שנית, במידה ויש תיקון באג, או בעיית אבטחת מידע לספריה שבשימוש, צריך לקמפל מחדש את הקובץ ולבנות אותו, במקום להצביע על הפונקציה המתוקנת בריצה שאחרי עדכון הספריה.
- בנוסף, הקובץ הוא מונוליטי, וככזה, אין לו יכולת להשתנות במידה והיו שינויים בספריות, אלא מה שקומפל אליו, הוא הדבר היחיד שהמערכת יכולה לתמוך בו.

קישור דינמי מאפשר לקחת הרבה מאוד פונקציות וספריות, ולטעון אותן בזמן ריצה. יש מספר יתרונות בנושא:

- תיקון בעיות בספריה, כאשר ה-ABI או API אינם משתנים, אינו דורש קימפול מחודש.
- היכולת לבצע Polymorphism עבור קוד שמיובא, כל עוד ה-API וה-ABI לא השתנו, ובכך לטעון תמיכה לפעולות שאותם רוצים לקבל, בהתאם למה שהספרייה עצמה מבצעת - דבר שנמצא לרוב בשימוש עם מונח שקיבל את השם: plugins.
- קובץ הריצה יהיה קטן יותר משמעותית מאשר קוד המקושר סטטית.

ישנן מספר חסרונות:

- התלות בגרסאות וABI ו-API זהים למה שקומפלה המערכת חשובה.
- יש הרבה ספריות בנוסף לקובץ הריצה אשר יטענו לזיכרון בזמן השימוש בהן.
- במידה וחסרה תלות מסוימת של ספריה, התכנה פשוט לא תרוץ.

תצורות ריצה

עכשיו שהדרך להגיע לתצורות ריצה ברורה יותר, אסביר מה הן תצורות ריצה. כאשר אני מדבר על תצורות ריצה, אני מדבר על קבצים כדוגמת exe. אז מדוע בעצם, לתת את השם "תצורות ריצה", ולא קבצי ריצה? בפרק זה אסביר לעומק את המצב. ישנן הרבה גישות כיצד קוד בסופו של דבר צריך לרוץ. יתרה מזאת, חלק מגישות אלו, מאפשרות לטעון קבצי משנה (ספריות עליהן דיברתי בקצרה בנושא המקשר), וכאלו אשר אינן יכולות לעשות את זה.

ישנן גישות אשר אומרות כי הקוד עצמו צריך לנהל הכל בכוחות עצמו, כולל מה שהיינו מצפים שמערכת ההפעלה תעשה, וכאלו אשר תלויים במימוש של מערכת הפעלה. אפילו סיומות, כדוגמת exe מטעות, היות וסיומת לשם של קובץ, אינו אומר מה המבנה של אותו הקובץ בפועל.

היות והנושא מורכב כל כך, להגיד "קבצי ריצה", אינו מכסה באמת את כל המקרים, ולכן למעשה השתמשתי במונח "תצורות ריצה" במקום.

הרעיון של תצורות ריצה

מרבית מערכות ההפעלה כיום, תומכות בהרצה של מספר רב של סוגי קבצים, אך לא כולם נכנסים לקטגוריה של "תצורת ריצה". למשל קבצים בעולם היוניקס שהם קבצי טקסט, אבל עם סימן shabeng (!#) והרשאות ריצה, עדיין קבצי טקסט, פשוט מכילים בתוכם מידע כיצד להריץ את התוכן, למשל מכילים מידע האומר להריץ את רובי או פיתון.

מרבית תצורות הריצה, כן מכילות הגדרות בסגנון ה-shabeng, אך הן למעשה magic number אשר מציין את סוג הקובץ. הסיבה לכך, היא שתצורת ריצה היא למעשה קובץ בינרי אשר מכיל מידע של שפת מכונה במבנה מסוים. ישנם מבנים רבים כאלו, בחלק הבא של המאמר אתמקד במספר קטן שלהם, אשר נמצא בשימוש העיקרי בעולם המחשבים כיום ואסביר אותם לעומק.

תתי תצורות ריצה

לפני שאסביר על סוגי התצורה עצמם, חשוב להבין כי לא בכל מצב בו יוצרים תצורת ריצה, ניתן ממש להריץ את הקובץ עצמו. לתצורות הריצה, יש יכולת להגדיר צורות בהן ניתן להשתמש בתוצר בעוד שימושים, כדוגמת קובץ אובייקט, אשר יהפוך לאחד מהמבנים הבאים:

- קובץ ריצה
- ספרייה משותפת
- ספרייה סטטית

בנוסף, ניתן להכניס לקבצים אלו גם מידע שמסייע בדיבוג, ואף ניתן גם להגדיר למקשר להפריד בין קובץ הריצה/ספרייה משותפת, לבין מידע שמסייע לדיבוג, ובכך ליצור קבצים נפרדים שיטענו רק כאשר רוצים לדבג את הריצה של המערכת.

ישנם פורמטים של קבצי ריצה, אשר מאפשרים להכניס גם metadata ואפילו לבצע פעולות embedded של מספר קבצים לאחד, ובכך למשל לשלב תמונות בתוך קובץ הריצה עצמו, אותם ניתן למצוא במיקום מיועד.

בנוסף, חשוב להבין, כי בפורמטים אשר מסוגלים ליצור ולטעון ספרייה משותפת, תצורת הריצה זהה לתוצר שיהיה לקובץ הריצה, אך עם התנהגות שונה. למשל, קובץ ריצה, מכיל כתובות קבועות עבור symbols והקריאות שלו, בעוד שבספרייה משותפת, מופעלת יכולת אשר נקראת PIC - Position

Independent Code, וזו מאפשרת לקבוע שטעינת הפונקציות מהספרייה המשותפת, תוכל לקבל את טווח הכתובות של קובץ הריצה מבלי לדרוס כתובות קיימות, כך שכל קובץ ריצה, יכיל מרחב כתובות מעט שונה בזמן הטעינה שלו, וכלל הספריות יהיו חלק מאותן הכתובות.

בחלק הבא של מאמר זה, אסביר לעומק על מבני קבצי הריצה השונים.

סיכום

בחלק זה התחלתי להיכנס לעולם של קבצי ריצה. הסברתי מה הם קבצי ריצה טבעיים ומה התפקיד שלהם. הסברתי כי מדובר למעשה ב"תצורות ריצה", ומדוע מומלץ לא להשתמש במונח "קבצי ריצה".

בחלק הבא אסביר לעומק על מספר מצומצם של תצורות ריצה, וכיצד בעצם הם עובדים בפועל.

מילון מונחים

- **API**, פירוש: Application Programming Interface. דרך לספק פונקציות לשימוש חוזר.
- **ABI**, פירוש: Application Binary Interface. חתימת זיכרון למבנה של פונקציה, שהקריאה אליה, וכן מבנה וסדר הנתונים שלה קבוע.

ישנן שיטות סידור שונות לפרמטרים שונים של פונקציות, והן תלויות במגוון הגדרות, כדוגמת מערכת הפעלה, האם מעבד משתמש ב-Big או Little Endian, ואפילו בהגדרה שניתנה לקומפילר לבצע. הסבר מפורט על סדר הקריאה של פרמטרים, ניתן למצוא בקישור הבא:
https://en.wikipedia.org/wiki/X86_calling_conventions