



دانشگاه صنعتی شریف
دانشکده‌ی مهندسی کامپیوتر

گزارش کار آزمایشگاه آزمایشگاه سیستم‌های عامل

گزارش آزمایش شماره ۶
(مدیریت حافظه)

۲۰

شماره‌ی گروه:

ارشیا یوسف‌نیا (۴۰۱۱۱۰۴۱۵)

گروه:

محمدعارف زارع زاده (۴۰۱۱۰۶۰۱۷)

دکتر بیگی

استاد درس:

تابستان ۱۴۰۴

تاریخ:

فهرست مطالب

۱	شرح آزمایش	۱
۱	۱.۱ استفاده از فراخوانی‌های malloc و free	۱.۱
۱	۲.۱ مشاهده ی وضعیت حافظه ی پردازش ها	۲.۱
۳	۳.۱ اجزای حافظه ی یک پردازش	۳.۱
۳	۴.۱ اشتراک حافظه	۴.۱
۴	۵.۱ آدرس‌های بخش‌های مختلف پردازش	۵.۱

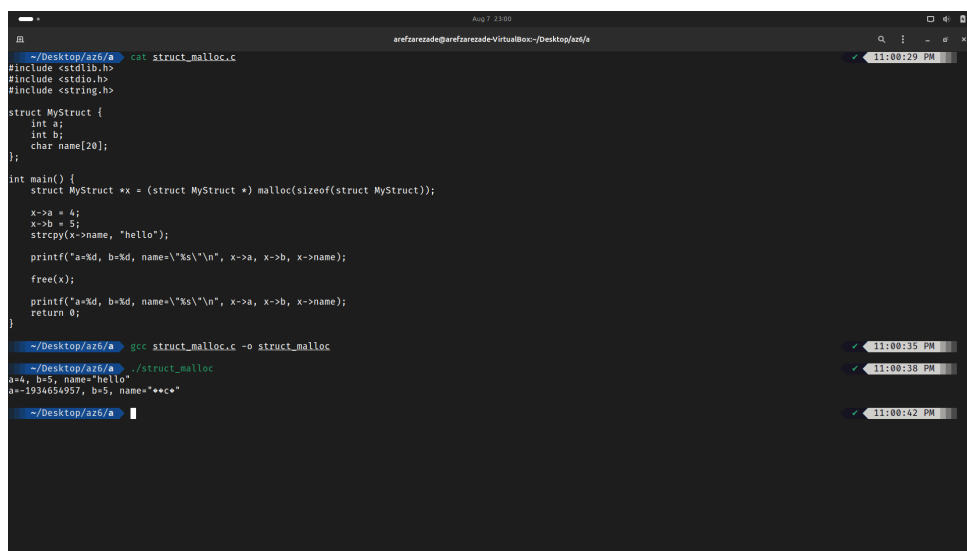
لیست تصاویر

۱	کد کار کردن با دستورات malloc و free	۱
۱	ستون user در دستور داده شده طبق صفحه‌ی man ps	۲
۲	ستون vsz در دستور داده شده طبق صفحه‌ی man ps	۳
۲	ستون rss در دستور داده شده طبق صفحه‌ی man ps	۴
۲	ستون pmem در دستور داده شده طبق صفحه‌ی man ps	۵
۲	ستون fname در دستور داده شده طبق صفحه‌ی man ps	۶
۲	اجرای دستور ps با ستون‌های خواسته شده	۷
۳	محل قرارگیری دستور ls و حافظه‌ی اختصاص داده شده به اجزای آن	۸
۴	کتابخانه‌های مشترک مورد استفاده توسط دستور ls	۹
۴	کتابخانه‌های مشترک مورد استفاده توسط تعدادی دستور دیگر	۱۰
۵	کد انتهای صفحه‌ی man etext	۱۱
۶	تعداد دفعات درخواست 1KB با دستور malloc برای تغییر آدرس انتهای heap	۱۲
۶	مشاهده‌ی رفتار بخش stack از حافظه	۱۳

۱ شرح آزمایش

۱.۱ استفاده از فراخوانی‌های malloc و free

خروجی دستور malloc یک پوینتر از نوع * void است. این پوینتر، به آدرسی که دستور malloc به اندازه‌ی خواسته شده حافظه را آماده کرده اشاره می‌کند. با تغییر نوع پوینتر (یا همان casting) می‌توان نوع این پوینتر را به پوینتر مناسب تغییر داد و به این صورت فیلدهای یک struct را تغییر داد یا در صورت نیاز کارهای دیگری کرد. کد خواسته شده را در شکل ۱ می‌توانید مشاهده کنید. در این کد، ابتدا با دستور malloc به اندازه‌ی کافی حافظه را برای یک instance از آن struct آماده کرده و سپس به فیلدهای آن مقدار دلخواهی می‌دهیم. سپس با دستور free سعی می‌کنیم آن بخش از حافظه را آزاد کرده، و سپس سعی می‌کنیم مثل قبل، فیلدهای آن را پرینت کنیم. در دفعه‌ی دوم، یا باید به ارور خورده، یا باید مقدار فیلدها تغییر کند و مقادیری تصادفی داشته باشد.



```
~/Desktop/az6/a cat struct_malloc.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct MyStruct {
    int a;
    int b;
    char name[20];
};

int main() {
    struct MyStruct *x = (struct MyStruct *) malloc(sizeof(struct MyStruct));

    x->a = 4;
    x->b = 5;
    strcpy(x->name, "hello");

    printf("a=%d, b=%d, name=\"%s\"\n", x->a, x->b, x->name);

    free(x);

    printf("a=%d, b=%d, name=\"%s\"\n", x->a, x->b, x->name);
    return 0;
}

~/Desktop/az6/a gcc struct_malloc.c -o struct_malloc
~/Desktop/az6/a ./struct_malloc
a=4, b=5, name="hello"
a=1934654957, b=5, name="***"
~/Desktop/az6/a
```

شکل ۱: کد کار کردن با دستورات malloc و free

۲.۱ مشاهده‌ی وضعیت حافظه‌ی پردازنده‌ها

ابتدا با کمک دستور man ps، هر یک از ستون‌ها را پیدا کرده، و سپس معنای هرکدام را می‌نویسیم. در کنار هرکدام، یک تصویر از صفحه‌ی man مربوط به آن ستون را نشان می‌دهیم.

- ستون user: این ستون مطابق شکل ۲ نام کاربری که پردازنده از طرف او اجرا می‌شود را می‌نویسد. همانطور که از شکل ۷ می‌توان مشاهده کرد، اگر پردازنده از طرف سیستم اجرا می‌شود، مقدار آن root می‌باشد.

user	EUSER	effective user name. This will be the textual user ID, if it can be obtained and the field width permits, or a decimal representation otherwise. The n option can be used to force the decimal representation. (alias uname, user).
------	-------	---

شکل ۲: ستون user در دستور داده شده طبق صفحه‌ی man ps

- ستون vsz: این ستون مطابق شکل ۳ میزان حافظه‌ی مجازی مربوط به آن پردازنده را برحسب کیلوبایت بیان می‌کند.

vsz **VSZ** virtual memory size of the process in KiB (1024-byte units). Device mappings are currently excluded; this is subject to change. (alias **vszsize**).

شکل ۳: ستون vsz در دستور داده شده طبق صفحه‌ی man ps

- ستون TSS: این ستون مطابق شکل ۴ میزان حافظه‌ی فیزیکی که پردازش در حال حاضر استفاده می‌کند را نشان می‌دهد. واضح است که مقدار این ستون همواره کمتر یا مساوی ستون vsz باید باشد. این موضوع در شکل ۷ قابل مشاهده است.

rss **RSS** resident set size, the non-swapped physical memory that a task has used (in kilobytes). (alias **rsssize**, **rsz**).

شکل ۴: ستون rss در دستور داده شده طبق صفحه‌ی man ps

- ستون pmem: این ستون مطابق شکل ۵ درصدی از کل حافظه را که پردازش در حال حاضر دارد استفاده می‌کند را نشان می‌دهد. بدیهی است که مقدار آن باید متناسب با rss باشد، و این موضوع در شکل ۷ قابل مشاهده است.

%mem **%MEM** ratio of the process's resident set size to the physical memory on the machine, expressed as a percentage. (alias **pmem**).

شکل ۵: ستون pmem در دستور داده شده طبق صفحه‌ی man ps

- ستون fname: این ستون مطابق شکل ۶ ۸ بایت (که در اکثر اوقات ۸ حرف است) اول نام فایل executable مربوط به آن پردازش را می‌نویسد. به بیان دیگر، برنامه‌ای که اجرای آن باعث ایجاد این پردازش شده است را نشان می‌دهد.

fname **COMMAND** first 8 bytes of the base name of the process's executable file. The output in this column may contain spaces.

شکل ۶: ستون fname در دستور داده شده طبق صفحه‌ی man ps

همچنین در شکل ۷ می‌توانید یک نمونه از اجرای دستور داده شده را مشاهده کنید.

```

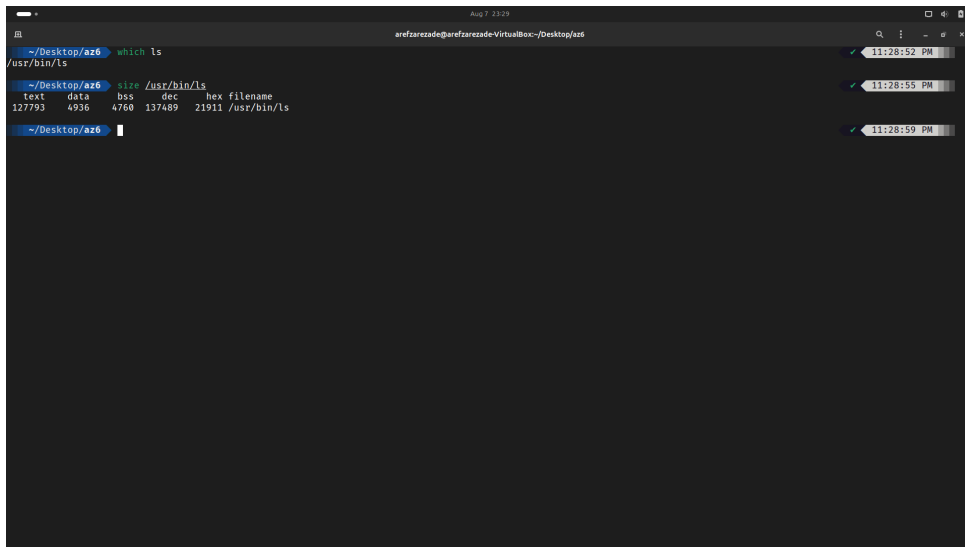
~/Desktop/az6  ps -o user,vsz,ss,pmem,fname -e | tail -n 40
arefzar+ 624260 80680 0.9 gsd-xset
arefzar+ 267224 24980 0.3 ibus-x11
arefzar+ 415884 26308 0.3 xdg-desk
arefzar+ 19488 1496 0.0 VBoxClie
arefzar+ 218296 4376 0.0 VBoxClie
arefzar+ 1102160 100620 1.2 mutter-x
arefzar+ 19488 1376 0.0 VBoxClie
arefzar+ 151592 2320 0.0 VBoxClie
root 479560 40512 0.4 fwupd
root 0 0 0.0 kworker/
arefzar+ 641604 33532 0.4 update-n
root 0 0 0.0 kworker/
root 0 0 0.0 psimon
root 0 0 0.0 kworker/
arefzar+ 1697264 295276 3.6 nautilus
arefzar+ 388324 8532 0.1 gvfsd-re
root 0 0 0.0 kworker/
root 0 0 0.0 kworker/
root 0 0 0.0 kworker/
root 0 0 0.0 kworker/
root 0 0 0.0 kworker/
arefzar+ 714380 63680 0.7 gnome-te
arefzar+ 18868 9984 0.1 zsh
arefzar+ 13532 4636 0.0 zsh
arefzar+ 14724 5476 0.0 zsh
arefzar+ 14708 5180 0.0 zsh
arefzar+ 2896 1560 0.0 glstatu
root 0 0 0.0 kworker/
root 0 0 0.0 kworker/
root 0 0 0.0 kworker/
root 0 0 0.0 kworker/
arefzar+ 2807928 63044 0.7 gjs
root 0 0 0.0 kworker/
root 0 0 0.0 kworker/
arefzar+ 400388 29620 0.3 tracker-
arefzar+ 13420 4140 0.0 ps
arefzar+ 8320 1892 0.0 tail
~/Desktop/az6

```

شکل ۷: اجرای دستور ps با ستون‌های خواسته شده

۳.۱ اجزای حافظه‌ی یک پردازنده

در شکل ۸ می‌توان محل قرارگیری دستور ls درون فایل سیستم و همچنین میزان تخصیص حافظه به بخش‌های گفته شده را مشاهده کرد.



```
~/Desktop/az6 ~$ which ls
/usr/bin/ls

~/Desktop/az6 ~$ size /usr/bin/ls
   text  data  bss   dec   hex filename
127793  4936  4760 137489 21911 /usr/bin/ls

~/Desktop/az6 ~$
```

شکل ۸: محل قرارگیری دستور ls و حافظه‌ی اختصاص داده شده به اجزای آن

همانطور که می‌توان مشاهده کرد، مقادیر data و text و bss را می‌توان مشاهده کرد. همچنین حافظه‌ی مربوط به initialized data با اینکه به طور مستقیم داده نشده است، با کمک اجزای دیگر می‌توان آن را به دست آورد (اختلاف data و bss).

اما مقادیر command-line arguments and environment variables (به خاطر متفاوت بودن در هربار اجرای دستور) و همچنین stack و heap (به خاطر متغیر بودن و همواره کم یا زیاد شدنشان حین اجرای دستور) هنگام اجرای دستور size گزارش نمی‌شوند.

۴.۱ اشتراک حافظه

در شکل ۹ می‌توان کتابخانه‌های مشترک مورد استفاده‌ی دستور ls را مشاهده کرد. همچنین در شکل ۱۰ کتابخانه‌های مشترک تعدادی دستور دیگر (nano و touch و size) را مشاهده کرد.

```

~/Desktop/az6 ~ which ls
/usr/bin/ls

~/Desktop/az6 ~ size /usr/bin/ls
text data bss dec hex filename
127793 4936 4760 137489 21911 /usr/bin/ls

~/Desktop/az6 ~ ldd /usr/bin/ls
linux-vdso.so.1 (0x000078f2e5b26000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x000078f2e5bc0000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x000078f2e5800000)
libpcre2-8.so.0 => /lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x000078f2e5b26000)
/lib64/ld-linux-x86-64.so.2 (0x000078f2e5c28000)

~/Desktop/az6 ~

```

شکل ۹: کتابخانه‌های مشترک مورد استفاده توسط دستور ls

```

~/Desktop/az6 ~ which nano
/usr/bin/nano

~/Desktop/az6 ~ ldd /usr/bin/nano
linux-vdso.so.1 (0x0000733ce977e000)
libncursesw.so.6 => /lib/x86_64-linux-gnu/libncursesw.so.6 (0x0000733ce96e9000)
libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x0000733ce96b5000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x0000733ce9400000)
/lib64/ld-linux-x86-64.so.2 (0x0000733ce9780000)

~/Desktop/az6 ~ which touch
/usr/bin/touch

~/Desktop/az6 ~ ldd /usr/bin/touch
linux-vdso.so.1 (0x00007ede30f5c000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ede30c00000)
/lib64/ld-linux-x86-64.so.2 (0x00007ede30f5e000)

~/Desktop/az6 ~ which size
/usr/bin/size

~/Desktop/az6 ~ ldd /usr/bin/size
linux-vdso.so.1 (0x0000753804130000)
libbfd-2.42-system.so => /lib/x86_64-linux-gnu/libbfd-2.42-system.so (0x0000753803fa0000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x0000753803c00000)
libz.so.1 => /lib/x86_64-linux-gnu/libz.so.1 (0x0000753803f93000)
libstdc++.so.1 => /lib/x86_64-linux-gnu/libstdc++.so.1 (0x0000753803cd9000)
libstdc++.so.1 => /lib/x86_64-linux-gnu/libstdc++.so.1 (0x0000753803cd9000)
/lib64/ld-linux-x86-64.so.2 (0x0000753804136000)

~/Desktop/az6 ~

```

شکل ۱۰: کتابخانه‌های مشترک مورد استفاده توسط تعدادی دستور دیگر

همانطور که می‌توان مشاهده کرد، بین اینها، بعضی کتابخانه‌ها مانند linux-vdso.so.1 در همه‌ی دستورات تست شده مورد استفاده قرار گرفته اند، و سیستم عامل به جای اینکه هربار جداگانه آنها را وارد حافظه کند، بهتر است یک بار وارد حافظه کرده و بین آنها بخش text را به اشتراک بگذارد.

۵.۱ آدرس‌های بخش‌های مختلف پرده

نکته: خواسته‌های این بخش در pdf داده شده و همچنین گیت‌هاب، تفاوت‌های جزئی دارد. طبق آخرین پیام کوئرا در زمان نوشتن گزارش، طبق صورت آزمایش موجود در گیت‌هاب به خواسته‌ها جواب می‌دهیم.

- کد گفته شده و همچنین اجرای آن را در شکل ۱۱ می‌توانید مشاهده کنید. همانطور که در این شکل می‌توان دید، مقدار etext که انتهای بخش text است از همه کوچکتر است. سپس مقدار edata که آدرس انتهای

بخش data که همان انتهای initialized data است کوچکتر است، و در نهایت مقدار end که همان آدرس انتهای بخش uninitialized data است از همه بزرگتر است. این موضوع با موارد گفته شده در صورت آزمایش در تطابق است.

```

~/Desktop/az6/e cat test_extern.c
#include <stdio.h>
#include <stdlib.h>

extern char etext, edata, end; /* The symbols must have some type,
                                or 'gcc -Wall' complains */

int
main(void)
{
    printf("First address past:\n");
    printf("  program text (etext)    %10p\n", &etext);
    printf("  initialized data (edata) %10p\n", &edata);
    printf("  uninitialized data (end)  %10p\n", &end);

    exit(EXIT_SUCCESS);
}

~/Desktop/az6/e gcc test_extern.c -o test_extern
~/Desktop/az6/e ./test_extern
First address past:
  program text (etext)    0x5ccb150e0211
  initialized data (edata) 0x5ccb150e3010
  uninitialized data (end) 0x5ccb150e3018
~/Desktop/az6/e

```

شکل ۱۱: کد انتهای صفحه‌ی man etext

دلیل اینکه در کامنت به اینها symbol گفته شده است، این است که اینها متغیر معمولی نیستند، بلکه labelهایی هستند که linker استفاده می‌کند تا انتهای بخش‌های گفته شده را تشخیص دهد. کلیدواژه‌ی extern در تعریف آنها استفاده می‌شود، تا به کامپایلر گفته شود که این نمادها در این کد وجود ندارند و به صورت خارجی مقدار آنها مشخص می‌شود. بنابراین کامپایلر می‌فهمد که لازم نیست مقدار یا آدرس اینها را بداند و می‌فهمد که linker بعداً مقدار مناسب را به آن می‌دهد. پس به جای مقدار دادن به آن، در object file آن را به عنوان نماد نگه می‌دارد. معنای کامنت موجود در کد هم این است که با اینکه این نمادها متغیر نیستند، اگر مانند متغیرها با آنها رفتار نشود، با اینکه از لحاظ syntax این کد درست است، در صورت فلگی مانند -Wall- حین کامپایل، هشدار داده می‌شود. با دادن type به آنها، این مشکل رفع می‌شود.

- در شکل ۱۲ می‌توان کد خواسته شده مربوط به تغییر آدرس انتهای heap را مشاهده کرد. کد زیر ابتدا یک بایت را با دستور malloc درخواست می‌کند. دلیل آن این است که اگر این کار را نکنیم، نتیجه‌ی خواسته شده را مشاهده نمی‌کنیم. دلیل آن هم با توضیحات پاراگراف بعدی واضح می‌شود. سپس در ادامه‌ی کد، در یک حلقه، آن قدر با دستور malloc مقدار 1KB حافظه از سیستم عامل درخواست می‌کنیم تا مقدار sbrk(0) که آدرس انتهای heap است تغییر کند. سپس نتایج را چاپ می‌کنیم. همانطور که می‌توان مشاهده کرد، این حلقه 130 بار اجرا شد تا آدرس انتهای heap تغییر کند.


```

~/Desktop/az6/e cat test_heap.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    void *start = malloc(1);

    void *init_heap = sbkr(0);

    int hops = 0;

    while (init_heap == sbkr(0)) {
        void *tmp = malloc(1024);
        hops++;
    }

    printf("Address of End of Heap at the start of the program: %p\n", init_heap);
    printf("Address of End of Heap at the end of the program: %p\n", sbkr(0));
    printf("Number of required 1KB blocks to increase the pointer at the end of heap: %d\n", hops);

    return 0;
}

~/Desktop/az6/e gcc test_heap.c -o test_heap
~/Desktop/az6/e ./test_heap
Address of End of Heap at the start of the program: 0x614add9f000
Address of End of Heap at the end of the program: 0x614addc000
Number of required 1KB blocks to increase the pointer at the end of heap: 130
~/Desktop/az6/e

```

شکل ۱۲: تعداد دفعات درخواست 1KB با دستور malloc برای تغییر آدرس انتهای heap

همانطور که می‌توان مشاهده کرد، باید چند بار مقدار کوچکی (مانند 1KB) را با دستور malloc رزرو کرد تا آدرس انتهای heap تغییر کند. دلیل آن این است که هربار اجرای فراخوانی سیستمی برای رزرو حافظه از لحاظ عملکرد غیر بهینه است، برای همین، دستور malloc در صورت نیاز به حافظه‌ی بیشتر، مقدار نسبتاً بزرگی را از سیستم عامل درخواست می‌کند، و با هربار اجرای malloc، بخشی از آن حافظه را بر برنامه می‌دهد. اینگونه تعداد دفعات درخواست حافظه از سیستم عامل به مراتب کمتر شده و عملکرد برنامه بهتر می‌شود.

- برای مشاهده‌ی تغییرات بخش stack، کد موجود در شکل ۱۳ را نوشته‌ایم. این کد، مطابق خواسته‌ی صورت آزمایش، دارای یک تابع بازگشتی است که یک متغیر به نام i ساخته، آدرس آن را چاپ کرده، و سپس دوباره خودش را اجرا می‌کند. همچنین اجرای این تابع را محدود کردیم که بیشتر از تعداد دفعات خواسته شده (که در این مورد 20 بار است) اجرا نشود.

```

~/Desktop/az6/e cat test_stack.c
#include <stdlib.h>
#include <stdio.h>

void rec_function(int repeats) {
    int i;
    printf("The address of i is: %p\n", &i);
    if (repeats > 0)
        rec_function(repeats - 1);
}

int main() {
    rec_function(20);
    return 0;
}

~/Desktop/az6/e gcc test_stack.c -o test_stack
~/Desktop/az6/e ./test_stack
The address of i is: 0x7ffdea652b84
The address of i is: 0x7ffdea652884
The address of i is: 0x7ffdea652584
The address of i is: 0x7ffdea652284
The address of i is: 0x7ffdea651f84
The address of i is: 0x7ffdea651c84
The address of i is: 0x7ffdea651984
The address of i is: 0x7ffdea651684
The address of i is: 0x7ffdea651384
The address of i is: 0x7ffdea651084
The address of i is: 0x7ffdea650d84
The address of i is: 0x7ffdea650a84
The address of i is: 0x7ffdea650784
The address of i is: 0x7ffdea650484
The address of i is: 0x7ffdea650184
The address of i is: 0x7ffdea64fe84
The address of i is: 0x7ffdea64fb84
The address of i is: 0x7ffdea64f884
The address of i is: 0x7ffdea64f584
The address of i is: 0x7ffdea64f284
The address of i is: 0x7ffdea64ef84
~/Desktop/az6/e

```

شکل ۱۳: مشاهده‌ی رفتار بخش stack از حافظه

همانطور که از خروجی کد بالا قابل مشاهده است، آدرس متغیر i هربار به مقدار ثابتی کاهش می‌یابد. این کاهش بیانگر این است که در هربار اجرای تابع، `stack` استفاده می‌شود (برای ذخیره‌ی `return address` و `stack pointer` قبلی و ...). سپس یک مقدار به اندازه‌ی 4 بایت به متغیر i اختصاص داده شده و روند تکرار می‌شود. همچنین اجرای این کد نشان می‌دهد که رشد `stack` رو به پایین (یا به سمت آدرس‌های کوچکتر) است.