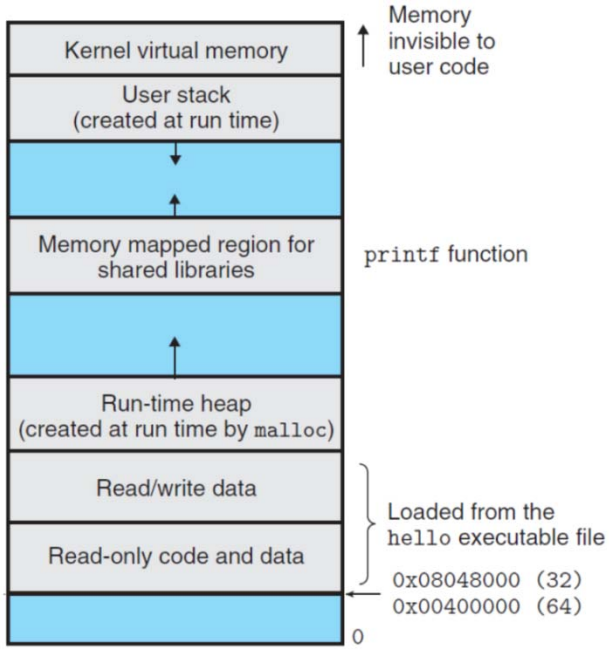
CSE320 System Fundamentals II

Function Call & Runtime Environment

YoungMin Kwon

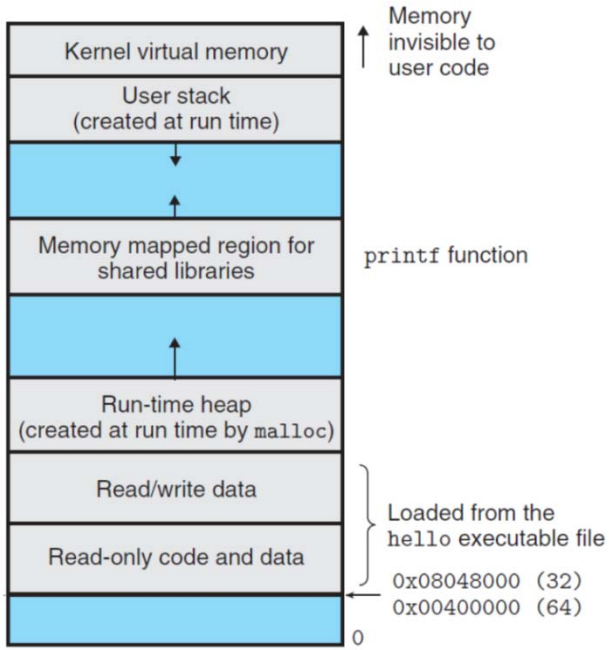
Storage Allocation

- .data
  - Readable, writable, initialized, global variables
- .rodata
  - Readonly, initialized, global variables
- .comm
  - Uninitialized global variables
  - Not allocated in the executable file. (mapped to a special file with 0s)
- .text
  - Instruction codes



Storage Allocation

- Stack
  - Local variables, parameters, temporary variables, …
  - Variable addresses are relative to rbp (stack frame pointer)
  - Top of the stack is marked by rsp
- Heap
  - Dynamic allocation (malloc)
  - Top of the heap is marked by brk
- Shared libraries
  - Library functions like printf

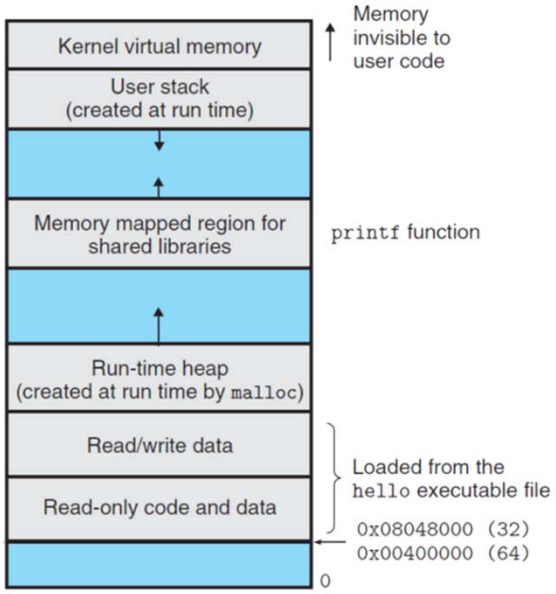


```c
int ei = 100;
static int si = 200;

int eu;
static int su;

const int ec = 300;
static const int sc = 400;

void foo(int p1, int p2) {
    extern void bar(char*);
    int ai = 500;
    int au;
    bar("Hello world");
}
```




```
        .globl  ei
        .data
ei:
        .long   100
si:
        .long   200
        .comm   eu,4,4 #(name,size,alignment) to bss
        .local  su
        .comm   su,4,4 # equiv to .lcomm su
        .globl  ec
        .section        .rodata
ec:
        .long   300
sc:
        .long   400
.LC0:
        .string "Hello world"
        .text
        .globl  foo
foo:
        pushq   %rbp      # caller's stack frame
        movq    %rsp, %rbp  # foo's stack frame
        subq    $32, %rsp   # alloc mem in stack
        movl    %edi, -20(%rbp) # p1, p1
        movl    %esi, -24(%rbp) # p2, p2
        movl    $500, -4(%rbp)  # int ai = 500;
…
        ret
```

Function Definition

- Functional Abstraction
  - Developers don't need to know the implementation details of a function
- Function Declaration
  - `type function_name(formal_parameter_list);`
- Function Definition
  - `type function_name(formal_parameter_list) function_body`

```
int foo(int a);   // function declaration
int bar(int a) {  // function definition
    return foo(a);
}
```

Function Call

- Function Call
  - `function_name(actual_parameter_list)`
  - Formal parameter variables are bound to the actual parameter values and the function body is executed in the new environment
  - Formal parameter variables can be used like local auto variables

```
int foo(int a) {
    int b = a;

    a = a + 1, b = b + 1;
    return a + b;
}
```

```
int bar(int b) {
    return foo(b);
}
```

Parameter Passing

- Call by Value
  - Expressions for actual parameters are evaluated and their values are copied to the corresponding formal parameter variables
- Actual parameters
  - Pushed to the stack from the caller and accessed by the callee
  - In GCC, the first 6 parameters are passed in registers for the performance:
    - `rdi, rsi, rdx, rcx, r8, r9` in this order

```
printf(
  "hello world %d, %d\n",
  g, a);
movl    g(%rip), %eax
movl    -4(%rbp), %edx
movl    %eax, %esi
leaq    .LC0(%rip), %rdi
movl    $0, %eax
call    printf@PLT
```

Parameter Passing

- Quiz: what will be the output of the following program?

```
#include <stdio.h>

void foo(int a, int b, int c) {
}

int main() {
    foo( printf("a\n"),
         printf("b\n"),
         printf("c\n") );
    return 0;
}
```

```
int foo(
    int a0, int a1, int a2, int a3,
    int a4, int a5, int a6, int a7,
    int a8, int a9, int aa, int ab,
    int ac, int ad, int ae, int af) {
    return a0 + a1 + a2 + a3 +
           a4 + a5 + a6 + a7 +
           a8 + a9 + aa + ab +
           ac + ad + ae + af;
}

int main() {
    foo(0,  1,  2,  3,
        4,  5,  6,  7,
        8,  9,  10, 11,
        12, 13, 14, 15);
    return 0;
}
```

```
main:
    pushq   %rbp            # save caller's stack frame pointer
    movq    %rsp, %rbp      # set up callee's stack frame pointer

    subq    $80, %rsp       # making space for actual parameters

    movl    $15, 72(%rsp)   # from the 7th param, passed through stack
    movl    $14, 64(%rsp)
    movl    $13, 56(%rsp)
    movl    $12, 48(%rsp)
    movl    $11, 40(%rsp)
    movl    $10, 32(%rsp)
    movl    $9,  24(%rsp)
    movl    $8,  16(%rsp)
    movl    $7,  8(%rsp)
    movl    $6,   (%rsp)

    movl    $5, %r9d        # the first 6 params are passed in registers
    movl    $4, %r8d
    movl    $3, %ecx
    movl    $2, %edx
    movl    $1, %esi
    movl    $0, %edi

    call    foo             # call the function foo

    leave                   # restore the caller's stack frame pointer
    ret                     # return to the caller
```

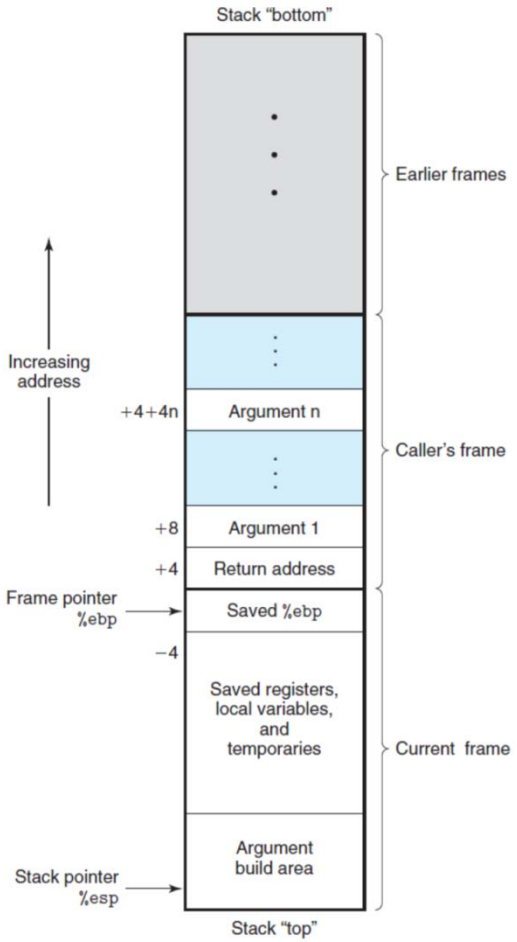
Stack Frame

- For each function call, a stack frame is generated and is pushed onto the stack

- A stack frame is composed of
  - Formal parameter variables
  - Return address
  - Auto variables for the function
  - Temporary variables
  - Registers to be preserved

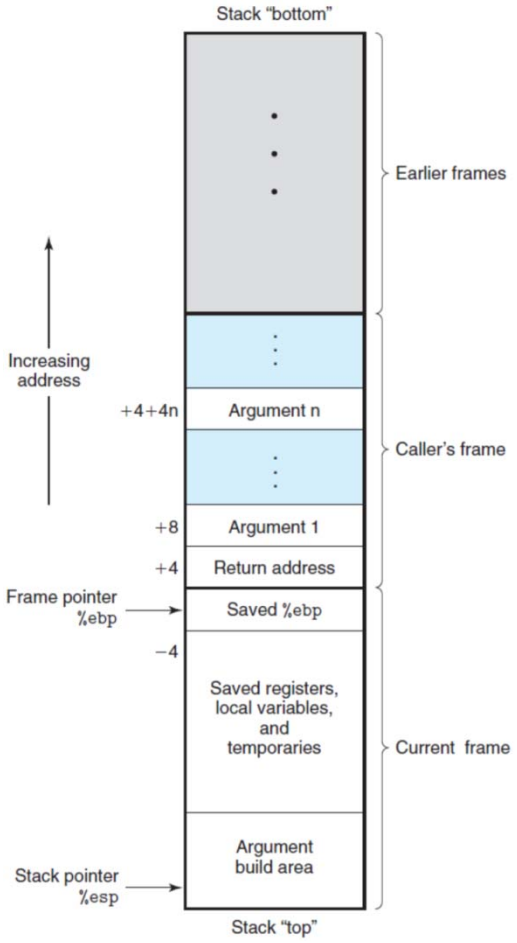
Stack Frame

- To make a function call
  - Caller pushes the actual parameters to the stack
  - Caller pushes the return address and jump to the function
    - call instruction pushes rip and jumps to the function



Stack Frame

- To make a function call
  - Callee saves caller's stack frame pointer (push rbp)
  - Callee sets callee's stack frame pointer (copy rsp to rbp)
  - Callee allocates space for its stack frame
    - decrease rsp by the amount of space for registers, local variables, and temporaries



```
int foo(
    int a0, int a1, int a2, int a3,
    int a4, int a5, int a6, int a7,
    int a8, int a9, int aa, int ab,
    int ac, int ad, int ae, int af)
{
    return a0 + a1 + a2 + a3 +
           a4 + a5 + a6 + a7 +
           a8 + a9 + aa + ab +
           ac + ad + ae + af;
}

int main()
{
    foo(0,  1,  2,  3,
        4,  5,  6,  7,
        8,  9,  10, 11,
        12, 13, 14, 15);
    return 0;
}
```

```
foo:
    pushq   %rbp            # save caller's stack frame pointer
    movq    %rsp, %rbp      # set up callee's stack frame pointer

    movl    %edi,  -4(%rbp) # copy the params in register to stack
    movl    %esi,  -8(%rbp)
    movl    %edx, -12(%rbp)
    movl    %ecx, -16(%rbp)
    movl    %r8d, -20(%rbp)
    movl    %r9d, -24(%rbp)

    movl    -4(%rbp), %edx  # accumulate the params to edx
    movl    -8(%rbp), %eax
    addl    %eax, %edx
    movl    -12(%rbp), %eax
    addl    %eax, %edx
    movl    -16(%rbp), %eax
    addl    %eax, %edx
…
    movl    88(%rbp), %eax

    addl    %edx, %eax      # return value of foo is in eax

    popq    %rbp            # restore the caller's stack frame pointer
    ret                     # return to the caller
```

Macros (#define)

- Function call
  - Evaluate expressions for the actual parameters in the reverse order,
  - Copy the results to the formal parameters,
  - Jump to the function body
- Macro expansion
  - Generate code by replacing the macro parameters with the provided expressions

```
#define PI 3.14
#define MAX(a,b) (a > b ? a : b)
MAX(PI,2) => (3.14 > 2 ? 3.14 : 2)

// Unexpected results
MAX(i++, j++) => (i++ > j++ ? i++ : j++)

// Code size can grow fast
MAX(MAX(1,2),MAX(3,4))
( (1>2?1:2) > (3>4?3:4) ? (1>2?1:2) : (3>4?3:4) )
```

```c
// common.h
//
#ifndef __COMMON__
#define __COMMON__
#include <stdio.h>
#include <stdlib.h>

#define ON_FALSE_GOTO(exp, label, msg) {\
    if(!(exp)) {\
        char *str = (char*)msg;\
        if(str && str[0] != '\0')\
            fprintf(stderr, "%s in file: %s, function: %s, line: %d\n",\
                str, __FILE__, __FUNCTION__, __LINE__);\
        goto label;\
    }\
}

#define ON_FALSE_EXIT(exp, msg) {\
    if(!(exp)) {\
        char *str = (char*)msg;\
        if(str && str[0] != '\0')\
            fprintf(stderr, "%s in file: %s, function: %s, line: %d\n",\
                str, __FILE__, __FUNCTION__, __LINE__);\
        exit(1);\
    }\
}

#endif
```

Ensure that the contents are copied only once when common.h is included multiple times

Ignore the next char (newline)

File name, Function name, Line number

Parameters to main

- argc: number of parameters
- argv: array of parameter strings
- envp: a null-terminated string array of shell variables in "name=value" format

```
#include <stdio.h>
#define SEP "----------------------------\n"
int main(int argc, char **argv, char **envp) {
    int i;
    printf(SEP "argc:%d\n", argc);

    printf(SEP "argv:\n", argc);
    for(i = 0; i < argc; i++)
        printf("%s\n", argv[i]);

    printf(SEP "envp:\n");
    for(i = 0; envp[i]; i++)
        printf("%s\n", envp[i]);
}
```

```
youngmin.kwon@momgoose:~/home/cse320/func_env$ ./a.out hello world
---------------------------------
argc:3
---------------------------------
argv:
./a.out
hello
world
---------------------------------
envp:
TERM=xterm
SHELL=/bin/bash
XDG_SESSION_COOKIE=967de2e693057b0c39eaed78000002c2-1489322486.369199-2128251071
SSH_CLIENT=10.1.1.3 14803 22
SSH_TTY=/dev/pts/0
USER=youngmin.kwon
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;...
MAIL=/var/mail/youngmin.kwon
PATH=/usr/local/pbs/bin:/usr/local/pbs/sbin:/usr/local/espicetools:/usr/local/...
PWD=/home/faculty/youngmin.kwon/home/cse320/func_env
LANG=en_US.UTF-8
SHLVL=1
HOME=/home/faculty/youngmin.kwon
...
```

gdb: Let's Debug

- Compile with -g option
  - `gcc -g debug_me.c`
- Run gdb with a.out
  - `gdb a.out`
- Break points
  - `break main, break eval_expr.c:18, info break, delete 1`
- Run
  - `run, finish, continue, step (s), next (n), quit`
- Examine variables
  - `bt, info frame, info locals, info args, info stack, info files`
  - `frame 1, up, down`
  - `print buf, print *buf, print buf->str[buf->index]`
  - `display *buf, info display, undisplay 1`

Assignment 4

- Download compiler_fun.zip and implement all TODOs
  - Run make to compile
- Finish the implementation
  - spl will print out an assembly code for a given program
  - Using the redirection, save the output as an assembly file
    - ./spl test_gcd_sub.txt > test_gcd_sub.s
  - On Linux, compile the assembly file using gcc and run it
    - gcc test_gcd_sub.s
    - ./a.out
- Due date: TBD
  - Upload the changed files to blackboard in a single zip file

Assignment 4

- Caller
  - Push actual parameters to the stack
    - We are not using registers
  - Call the function
  - Remove the parameters by adding to rsp
  - Push the result in rax to the stack
    - Evaluated value of an expression is at the top of the stack
- Callee
  - Manage stack frame pointers
  - Allocate space for local variables by subtracting from rsp
  - Pass the result in rax

Assignment 4

- Variable locations
  - Global variables
    - Allocated in .data area when the program is loaded
    - Address: (VAR + offset)(%rip)
  - Parameters
    - Allocated in the stack by the caller
    - Address: offset(%rbp) (positive offset)
  - Local variables
    - Allocated in the stack by the callee
    - Address: offset(%rbp) (negative offset)

```
typedef struct program {
    refobj_t ref;   //ref is at the beginning of program
    void   ( *exec  )(struct program *self);
    void   ( *print )(struct program *self);

    list_t *vars;   //global variables
    list_t *funcs;  //function definitions
} program_t;

typedef struct func_list_entry {
    func_t *func;
    list_t lst;
} func_list_entry_t;

static void exec_program(program_t *self) {
…
    //TODO: generate header

    //translate function definitions
    for(list_t *pos = pgm->funcs->next; pos != pgm->funcs; pos = pos->next) {
        func_list_entry_t *entry = containerof(pos, func_list_entry_t, lst);
        //TODO: generate code for the function entry->func
        //hint: use entry->func->exec
    }

    //TODO: generate footer
…
}
```

```
    .global main
    .text

…

    .section .rodata
ANSW:       .string "answer: %ld\n"
ENTR:       .string "enter: "
ENTR_FMT:   .string "%ld"
    .data
VAR:        .zero 8
```

```
    .data
VAR:        .zero 8
```

```
static void footer() {
…
    if( var_store_local_var_size(NULL) > 0) {
        //TODO: allocate global variables
        //hint: these global variables are in .data section
        //hint: use VAR label for the variable name
        //hint: use .zero x to allocate x bytes
        //hint: use var_store_local_var_size(NULL)
    }
}


//function definitions
//
typedef struct func {
    refobj_t ref;   //ref is at the beginning of stmt
    void   ( *exec  )(struct func *self);
    void   ( *print )(struct func *self);

    char *fname;    //name of the function
    list_t *ids;    //names of the formal parameters
    stmt_t *body;   //body
} func_t;
```

```
static void exec_func(func_t *self) {
…
    //create a local_store for this function
    var_store_t store;
    var_store_init(&store, list_size(func->ids));
…

    //TODO: generate the preamble for the function
    //   1. add a label with the function name (hint: use func->fname)
    //   2. save the caller's stack frame pointer
    //   3. set the callee's stack frame pointer
    //   4. allocate stack space for local variables
    //      (hint: use var_store_local_var_size)

    //TODO: generate the body of the function
    //hint: use func->body->exec

    //TODO: generate the postamble for the function
    //hint: leave then return

    //destroy the local_store for this function
    var_store_destroy(&store);
}
```

```
    # fun sum(n)
sum:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
…
    # exit sum
    leave
    ret
```

```
typedef struct stmt_return {
…
    expr_t *expr;
} stmt_return_t;

static void exec_return(stmt_t *self, var_store_t *store) {
    ON_FALSE_EXIT(self->ref.tag == OBJ_STMT_RETURN,
                  strmsg("tag (%d) is not OBJ_STMT_RETURN", self->ref.tag));
…
    //TODO: evalate stmt->expr
    //hint: use stmt->expr->eval

    COMMENT("return ", stmt->expr->print(stmt->expr), "");

    //TODO: pop the result of stmt->expr to rax
    //hint: the result is in the stack


    //TODO: deallocate the stack space for local variables without using leave
    //hint: rbp has the stack pointer value before the allocation


    //TODO: restore the caller's stack frame pointer without using leave


    //TODO: return from this function
}
```

```
# return s
popq    %rax
movq    %rbp, %rsp
popq    %rbp

ret
```

```
typedef struct expr_func_app {
…
    char *fname;    //function name
    list_t *args;   //actual parameters of expr_list_entry_t list
} expr_func_app_t;

static void eval_func_app(expr_t *self, var_store_t *store) {
…
    //evaluate the actual parameters in the reverse order and
    //push the results to the stack
    for(list_t *pos = expr->args->prev ; pos != expr->args; pos = pos->prev) {
        expr_list_entry_t *arg = containerof(pos, expr_list_entry_t, lst);
        //TODO: evaluate the actual parameter
        //hint: use arg->expr->eval,
        //      this call will push the result onto the stack
    }

    //TODO: invoke the function
    //hint: use expr->fname

    //TODO: remove the actual parameters from the stack
    //hint: use list_size(expr->args)

    //TODO: push the result to the stack
    //hint: the result is in rax
}
```

```
# sum(10)
pushq   $10
call    sum
addq    $8, %rsp
push    %rax
```

```
typedef struct expr_var {
…
    char *id;       //variable
} expr_var_t;

static void eval_var(expr_t *self, var_store_t *store) {
…
    if(!var_store_has(store, expr->id))
        store = NULL;   //global variable
    int inx = var_store_get(store, expr->id); //get the index of the variable

    if(store == NULL) {
        //TODO: handle as a global variable
        //inx is a negative number. making it positive,
        //variable address is VAR - inx*8 from rip
    }
    else if(inx > 0) {
        //TODO: handle as a parameter
        //accounting for 16 bytes for return address and rbp and
        //accounting for 8 bytes for the push op (memory at rsp has data),
        //variable address is 16 + inx*8 - 8 from rbp
    }
    else {
        //TODO: handle as a local variable
        //accounting for 8 bytes for the push op (memory at rsp has data),
        //variable address is inx*8 - 8 from rbp
    }
}
```

```
# i (loc var)
movq    -8(%rbp), %rax
pushq   %rax

# n (param)
movq    16(%rbp), %rax
pushq   %rax
```

```
//program sum
var a
fun sum(n) {
    i := 0
    s := 0
    while( i <= n ) {
        s := s + i
        i := i + 1
    }
    return s
}
fun main() {
    a := sum(10)
    write a
}
```

```
    .globl main
    .text
…
```

```
    # fun main()
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $0, %rsp

    # 10
    pushq   $10

    # sum(10)
    call    sum
    addq    $8, %rsp
    push    %rax

    # a := sum(10)
    popq    %rax
    movq    %rax, (VAR + 0)(%rip)

    # a
    movq    (VAR + 0)(%rip), %rax
    pushq   %rax
    # write a
    call    write_fun
    addq    $8, %rsp

    # exit main
    leave
    ret
```

```
    # fun sum(n)
sum:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp

    # 0
    pushq   $0
    # i := 0
    popq    %rax
    movq    %rax, -8(%rbp)

    # 0
    pushq   $0
    # s := 0
    popq    %rax
    movq    %rax, -16(%rbp)

    # while ( i <= n ) ...
label_000:
    # i
    movq    -8(%rbp), %rax
    pushq   %rax

    # n
    movq    16(%rbp), %rax
    pushq   %rax

    # end while ( i <= n ) ...
label_001:

    # s
    movq    -16(%rbp), %rax
    pushq   %rax

    # return s
    popq    %rax

    movq    %rbp, %rsp

    popq    %rbp

    ret

    # exit sum
    leave
    ret
…
    .section .rodata
ANSW:       .string "answer: %ld\n"
ENTR:       .string "enter: "
ENTR_FMT:   .string "%ld"
    .data
VAR:        .zero 8
```