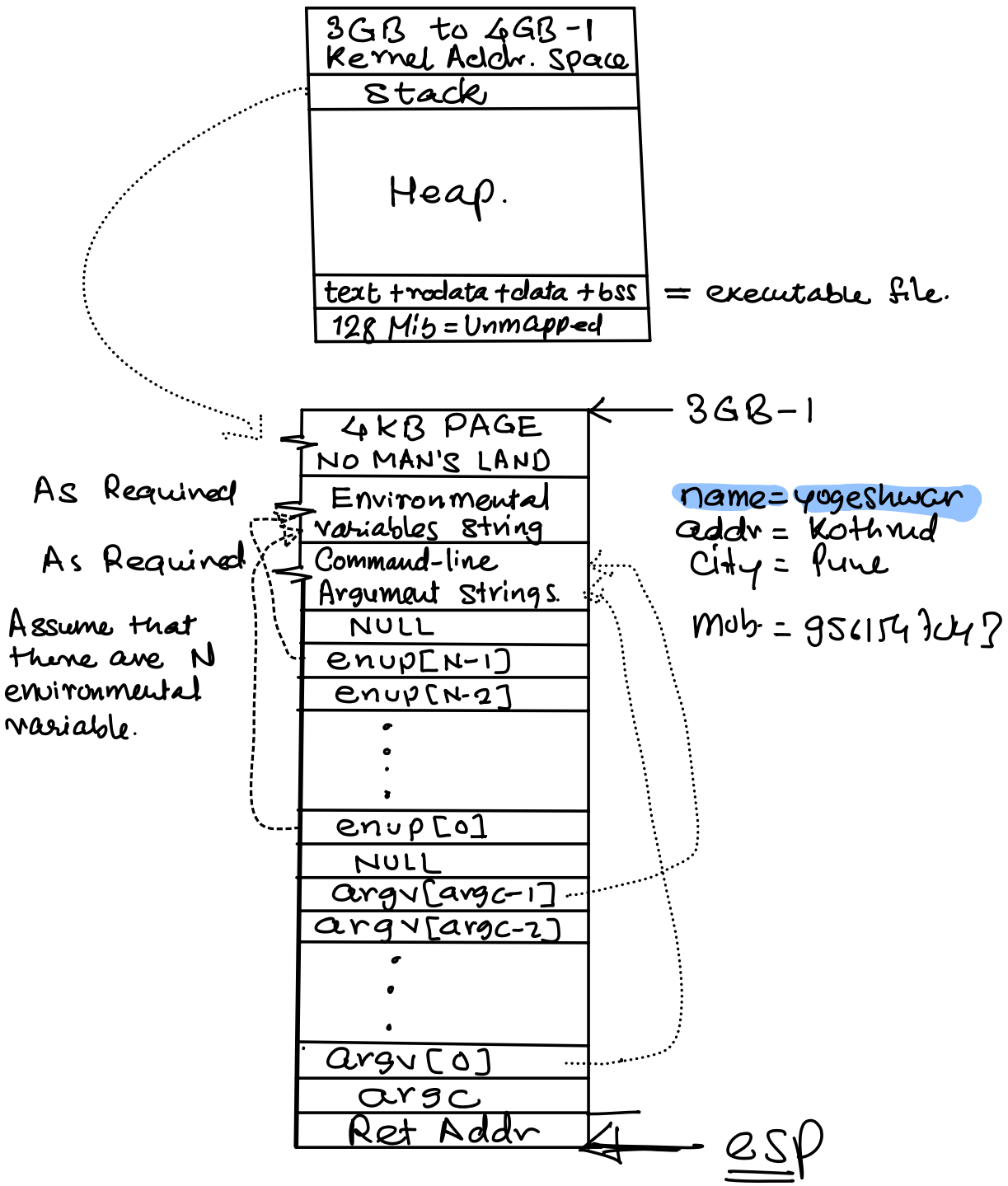
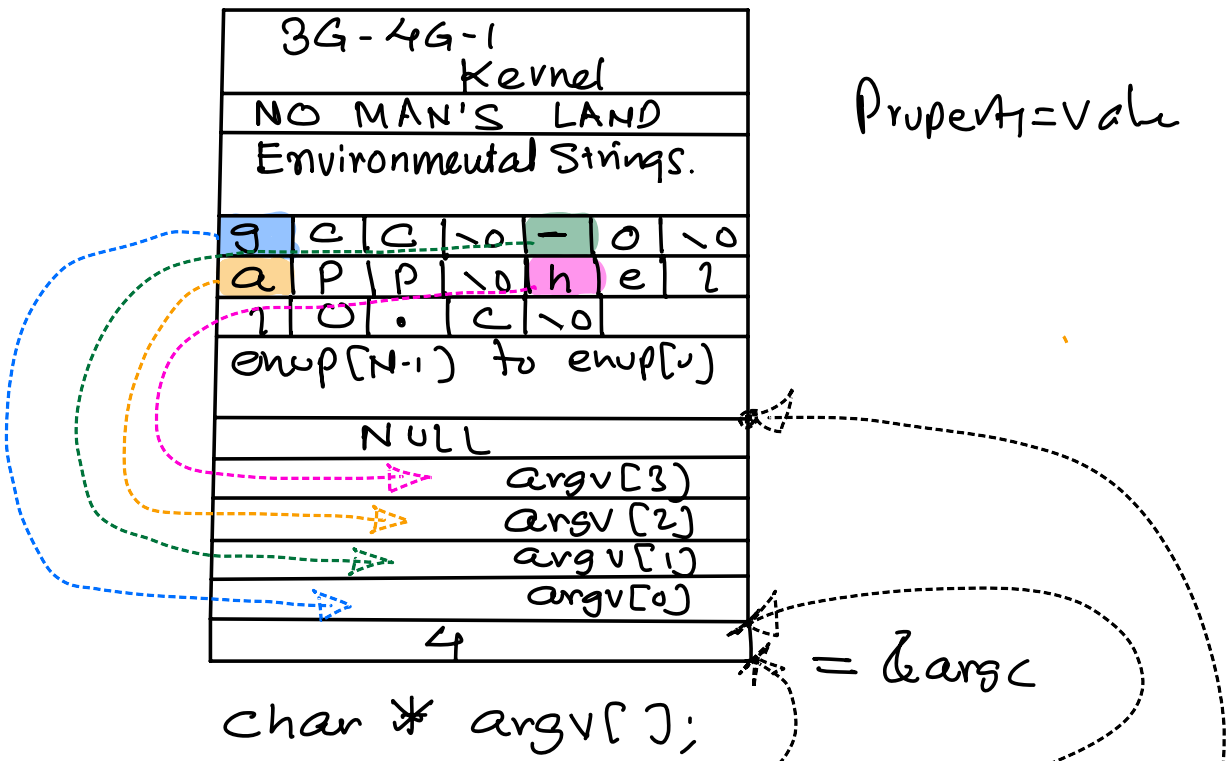
We have to learn the following instructions before we can understand the role of the prologue and epilogue in implementing the C calling convention.

1) push 2) pop 3) call 4) ret

For understanding the push and the pop instruction it is necessary that we visit the program stack once again.



#gcc -o app hello.c



char * argv[];

int main(int argc, char* argv[],
char* envp[])

int main(void) {

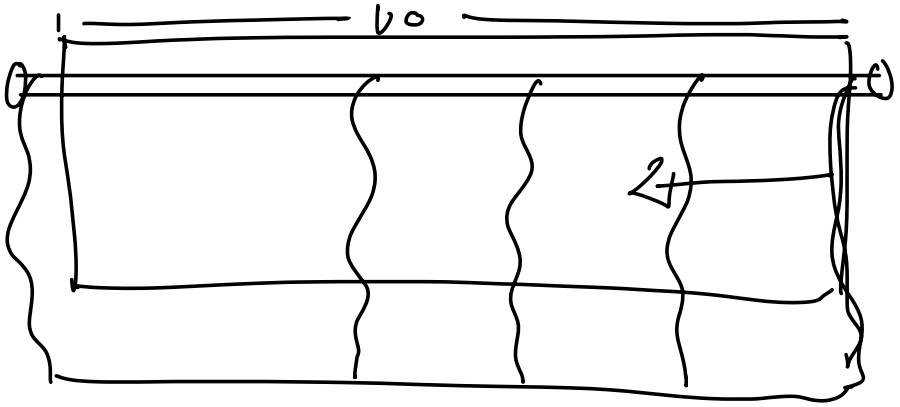
}

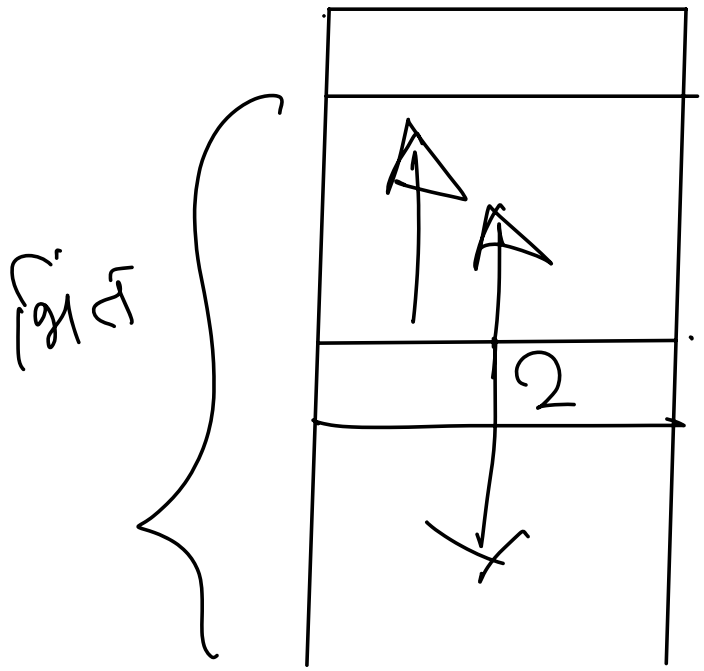
int main(int argc, char* argv[]) {

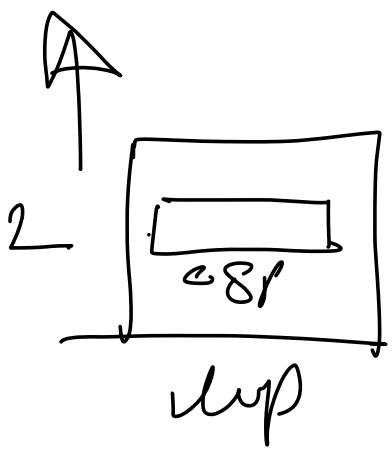
}

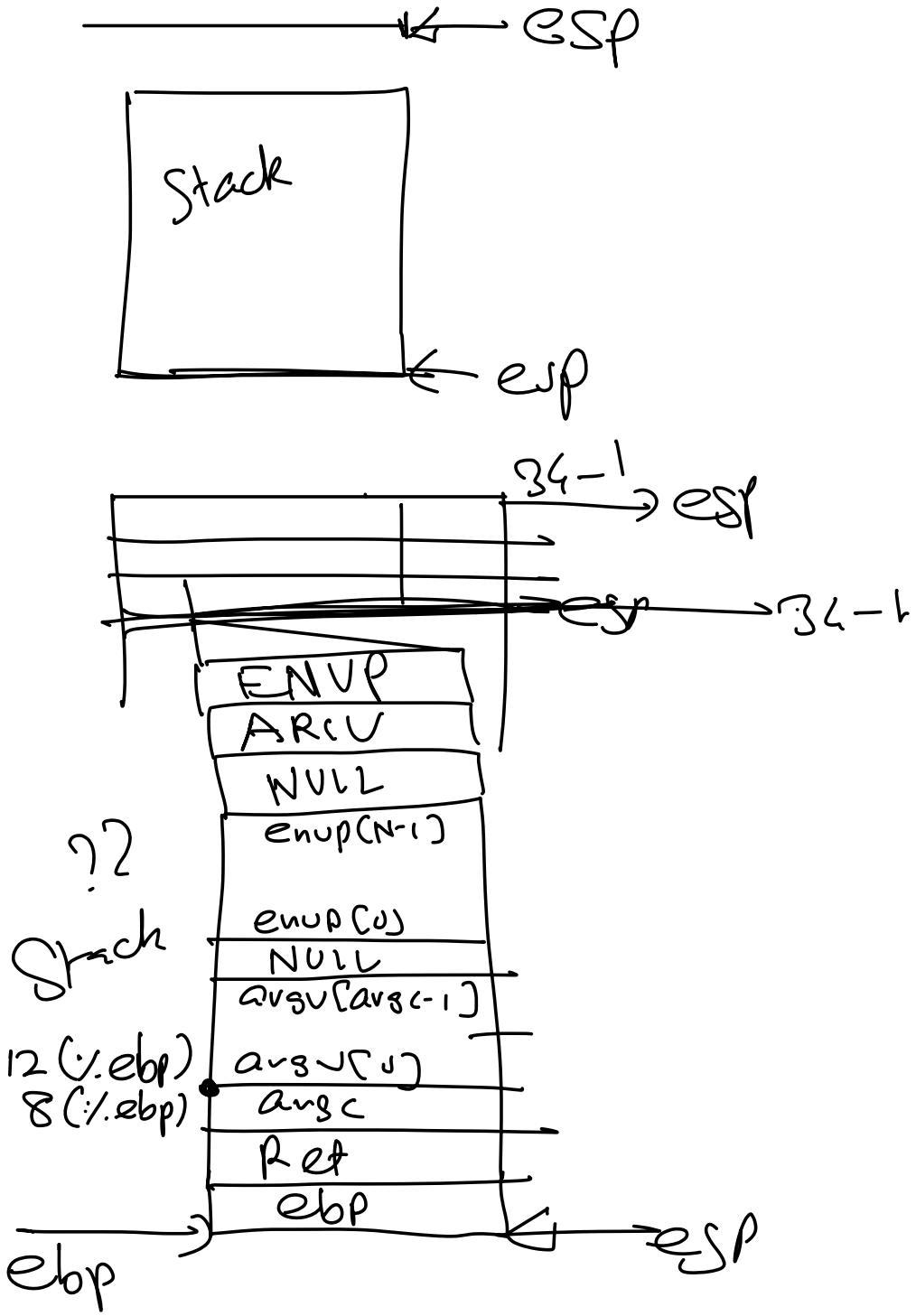
int main(int argc, char* argv[], char* envp[]) {

}


vo
2


2
Mem


2
esr
up



ebp + 12 argv

argv[2]

mukl $(2), %eax

movl 12(%ebp, %eax, 4), %edx

12+ebp+eax*4

12+ebp = argv

eax = index

4 = sizeof pm

12+ebp+eax*4.

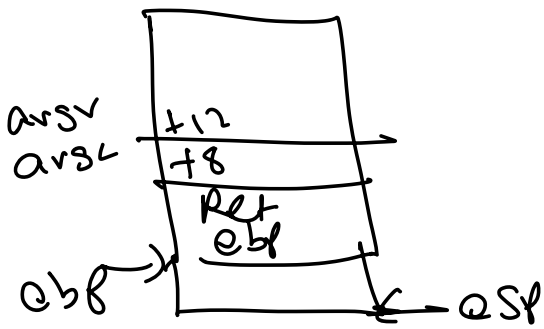
12(%ebp, %eax, 4)

(1) to argc-1

movl 12(%ebp,%eax,4),%edx

A program to print command line argument

```
.section .text
.globl main
.type main, @function
.equ   argc, 8
.equ   argv, 12
.equ   i, -4
main:
    pushl  %ebp
    movl   %esp, %ebp
    subl   $4, %esp

    movl   $0, -4(%ebp)  # i <- 0
    jmp    mn_cond_1
mn_for_1:
    # printf("argv[%d]:%s\n", i, argv[i]);
    movl   12(%ebp, %eax, 4), %edx    # edx <- M[x:x+3] where
                                      # x == 12+ebp+eax*4
    pushl  %edx
    pushl  %eax
    pushl  $msg_p1
    call   printf
    addl   $12, %esp

    addl   $1, -4(%ebp) # i <- i + 1
mn_cond_1:
    movl   -4(%ebp), %eax   # eax <- i
    cmpl   8(%ebp), %eax    # eax i.e. i is being compare with argc i.e. ebp+8
    jl     mn_for_1         # jmp to loop body if eax (i.e. i) is less than
                            # ebp + 8 (i.e. argc)

    pushl  $0
    call   exit


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



1) The push instruction:
   Syntax of push instruction.

   push(b/w/l) imm8/16/32

   push(b/w/l) reg8/16/32

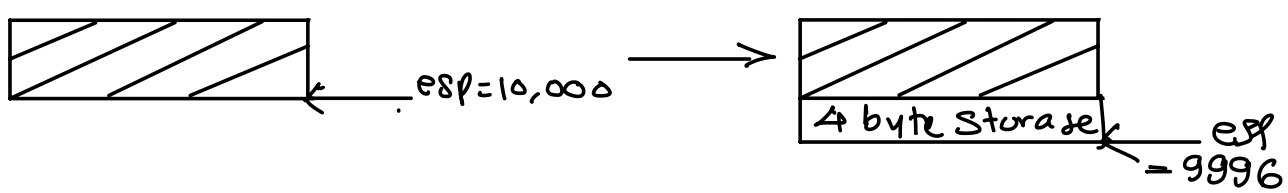
   push(b/w/l) memory_addr_in_direct_mode

Let us assume that num is a global variable.
say in BSS

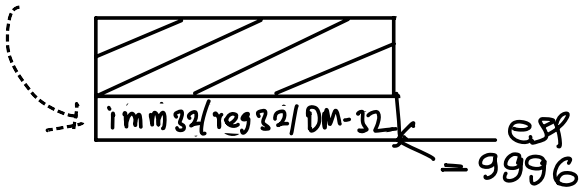
```
.section .bss
    .comm  num, 4, 4

pushl  num    # okay, because address of num is specified using direct addressing
              # mode

pushl  -4(%ebp)  # not okay, because push instruction does not accept register
                 # indirect addressing mode

pushl  $100 # okay
or
pushl  %edx   # okay
or
push   num    # okay
#-----------------------

pushl  imm32/reg32/direct_mem_32
```

Internal steps that are executed as a part of hardware implementation of push instruction:

Step 1: Decrement stack pointer (esp register) by 4 bytes (suffix 'l' == 4)
        creating 4 additional bytes of storage on stack.



Step 2: Move the operand of instruction (imm32,reg32,direct_mem_32) in
        the allocated storage



2) The pop instruction:
   Syntax:
   popl   %reg

Following steps are executed as hardware implementation of pop
1) Let current address in esp be x
   reg <- M[x : x + 3]
2) esp <- esp + 4 (x + 4)
   thereby freeing 4 bytes on stack

#----------------------------

HINT: after push instruction the esp has reduced by 4 and after pop instruction the esp has incremented by 4.

#----------------------------

3) call   addr (in text section)

   Direct addressing mode:

   call   function_name

   Indirect addressing mode:
   assuming address of function is in a register r

   call *%r

call   base_addr_of_function

Following steps are implemented by the hardware as a part of execution of the call instruction.

Step 1:   Compute the address of next instruction (next instruction of call instruction)
          Address of 'call' instruction is in eip register as 'call' is the current instruction.

addr of next instruction of 'call' instruction =
eip + sizeof(call instruction) -> obtained as a feedback from decode unit
We will call this address as 'RETURN ADDRESS' (because when block of instruction called by the 'call' instruction gets over, it should restore the control flow to this address)

Step 2: push the RETURN address on the stack.
esp <- esp - 4
M[esp : esp + 3] <- RETURN ADDRESS

Step 3: Set eip to operand address.
eip <- base_addr_of_function (callee functio)