Compilation, Linking, Loading

Abhijit A M

Review of last few lectures

Boot sequence: BIOS, boot-loader, kernel

Boot sequence: Process world

kernel->init -> many forks+execs() -> ....

Hardware interrupts, system calls, exceptions

Event driven kernel

System calls

Fork, exec, ... open, read, ...

What are compiler, assembler, linker and loader, and C library

System Programs/Utilities

Most essential to make a kernel really usable

Standard C Library

A collection of some of the most frequently needed functions for C programs

scanf, printf, getchar, system-call wrappers (open, read, fork, exec, etc.), ...

An machine/object code file containing the machine code of all these functions

Not a source code! Neither a header file. More later.

Where is the C library on your computer?

/usr/lib/x86_64-linux-gnu/libc-2.31.so

Compiler

application program, which converts one (programming) language to another

Most typically compilers convert a high level language like C, C++, etc. to Machine code language

E.g. GCC /usr/bin/gcc

Usage: e.g.

$ gcc main.c -o main

Here main.c is the C code, and "main" is the object/machine code file generated

Input is a file and output is also a file.

Other examples: g++ (for C++), javac (for java)

Assembler

application program, converts assembly code into machine code

What is assembly language?

Human readable machine code language.

E.g. x86 assembly code

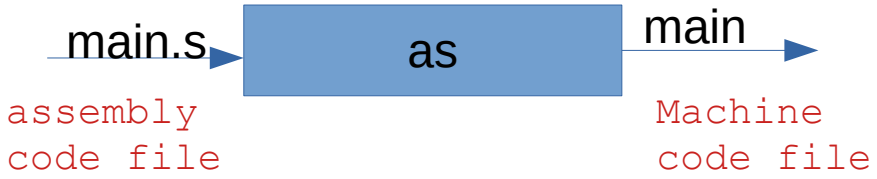
mov 50, r1

add 10, r1

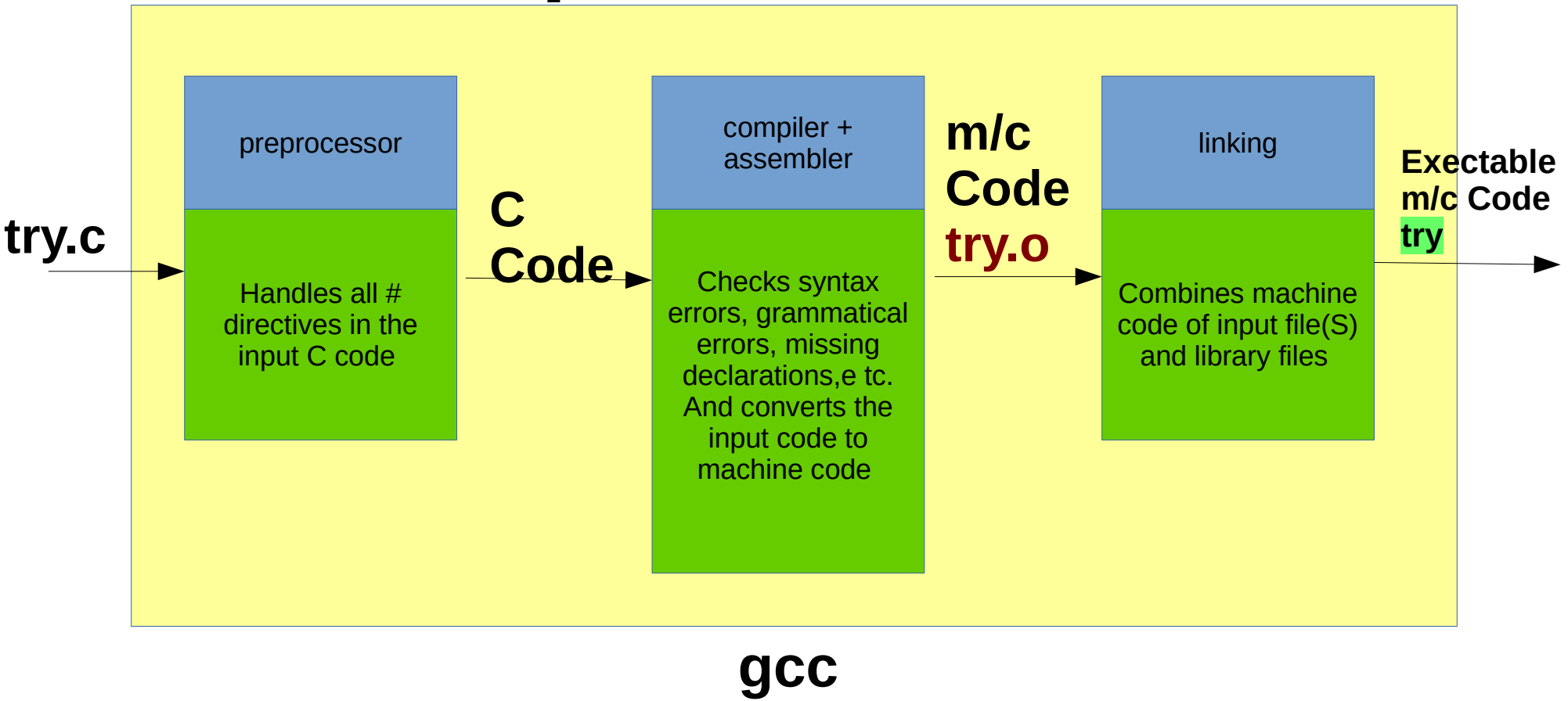
mov r1, 500

Usage. eg..

$ as something.s -o something



Compilation Process



Example

try.c

```
#include <stdio.h>
#define MAX 30
int f(int, int);
int main() {
        int i, j, k;
        scanf("%d%d", &i, &j);
        k = f(i, j) + MAX;
        printf("%d\n", k);
        return 0;
}
```

f.c

```
int g(int);
#define ADD(a, b) (a + b)
int f(int m, int n) {
        return ADD(m,n) + g(10);
}
```

g.c

```
int g(int x) {
        return x + 10;
}
```

Try these commands, observe the output/errors/warnings, and try to understand what is happening

```
$ gcc try.c
$ gcc -c try.c
$ gcc -c f.c
$ gcc -c g.c
$ gcc try.o f.o g.o -o try
$ gcc -E try.c
$ gcc -E f.c
```

More about the steps

Pre-processor

#define ABC XYZ

cut ABC and paste XYZ

include <stdio.h>

copy-paste the file stdio.h

There is no CODE in stdio.h, only typedefs, #includes, #define, #ifdef, etc.

Linking

Normally links with the standard C-library by default

To link with other libraries, use the -l option of gcc

Using gcc itself to understand the process

Run only the preprocessor

cc -E test.c

Shows the output on the screen

Run only till compilation (no linking)

cc -c test.c

Generates the “test.o” file , runs compilation + assembler

gcc -S main.c

One step before machine code generation, stops at assembly code

Combine multiple .o files (only linking part)

Linking process

Linker is an application program

On linux, it's the "ld" program

E.g. you can run commands like $ ld a.o b.o -o c.o

Normally you have to specify some options to ld to get a proper executable file.

When you run gcc

$ cc main.o f.o g.o -o try

the CC will internally invoke "ld" . ld does the job of linking

The resultatnt file "try" here, will contain the codes of all the functions and linkages also.

What is linking?

"connecting" the call of a function with the code of the function.

What happens with the code of printf()?

Executable file format

An executable file needs to execute in an environment created by OS and on a particular processor

Contains machine code + other information for OS

Need for a structured-way of storing machine code in it

Different OS demand different formats

Windows: PE, Linux: ELF, Old Unixes: a.out, etc.

ELF : The format on Linux.

Try this

$ file */bin/ls*

$ file /usr/lib/x86_64-linux-gnu/libc-2.31.so

Exec() and ELF

When you run a program

$ ./try

Essentially there willl be a fork() and exec("./try", ...")

So the kernel has to read the file "./try" and understand it.

So each kernel will demand it's own object code file format.

Hence ELF, EXE, etc. Formats

ELF is used not only for executable (complete machine code) programs, but also for partially compiled files e.g. main.o and library files like libc.so.6

What is a.out?

"a.out" was the name of a format used on earleir Unixes.

It so happened that the early compiler writers, also created executable with default name 'a.out'

Utilities to play with object code files

objdump

$ objdump -D -x /bin/ls

Shows all disassembled machine instructions and “headers”

hexdump

$ hexdump /*bin/*ls

Just shows the file in hexadecimal

readelf

Alternative to objdump

ar

To create a “statically linked” library file

$ ar -crs libmine.a one.o two.o

Gcc to create shared library

$ gcc hello.o -shared -o libhello.so

To see how gcc invokes as, ld, etc; do this

$ gcc -v hello.c -o hello

/* https://stackoverflow.com/questions/1170809/how-to-get-gcc-linker-command */

Linker, Loader, Link-Loader

Linker or linkage-editor or link-editor

The “ld” program. Does linking.

Loader

The exec(). It loads an executable in the memory.

Link-Loader

Often the linker is called link-loader in literature. Because where were days when the linker and loader’s jobs were quite over-lapping.

Static, dynamic / linking, loading

Both linking and loading can be

Static or dynamic

More about this when we learn memory management

An important fundamental:

memory management features of processor, memory management architecture of kernel, executable/object-code file format, output of linker and job of loader, are all interdependent and in-separable.

They all should fit into each other to make a system work

That's why the phrase "system programs"

Cross-compiler

Compiler on system-A, but generate object-code file for system-B (target system)

- E.g. compile on Ubuntu, but create an EXE for windows

Normally used when there is no compiler available on target system

- see gcc -m option

See https://wiki.osdev.org/GCC_Cross-Compiler

Calling Convention

Abhijit A M

The need for calling convention

An essential task of the compiler

- Generates object code (file) for given source code (file)
- Processors provide simple features
  - Registers, machine instructions (add, mov, jmp, call, etc.), imp registers like stack-pointer, etc; ability to do byte/word sized operations
  - No notion of data types, functions, variables, etc.
- But languages like C provide high level features
  - Data types, variables, functions, recursion, etc
- Compiler needs to map the features of C into processor's features, and then generate machine code
- In reality, the language designers design a language feature only after answering the question of conversion into machine code

The need for calling convention

Examples of some of the challenges before the compiler

- “call” + “ret” does not make a C-function call!
  - A “call” instruction in processor simply does this
    - Pushes IP(that is PC) on stack + Jumps to given address
  - This is not like calling a C-function !
  - Unsolved problem: How to handle parameters, return value?
- Processor does not understand data types!
  - Although it has instructions for byte, word sized data and can differentiate between integers and reals (mov, movw, addl, addf, etc.)

Compiler and Machine code generation

Example, code inside a function

```
int a, b, c;
c = a + b;
```

What kind of code is generated by compiler for this?

```
sub  12, <esp>  #normally local variables are located on stack, make space
mov <location of a in memory>, r1 #location is on stack, e,g. -4(esp)
mov <location of b in memory>, r2
add r1, r2 # result in r1
mov r1, <location of c in memory>
```

Compiler and Machine code generation

Across function calls

```
int f(int m, n) {
    int x = m, y = n;
    return g(x, y);
}
int x(int a) {
    return g (a, a+ 1);
}
int g(int p, int q) {
    p = p * q + p;
    return p
}
```

g() may be called from f() or from x()

Sequence of function calls can NOT be predicted by compiler

Compiler has to generate machine code for each function assuming nothing about the caller

Compiler and Machine code generation

Machine code generation for functions

Mapping C language features to existing machine code instructions.

Typical examples

a =100 ; ==> mov instruction

a = b+ c; ==> mov, add instructions

while(a < 5) { j++; } ==> mov, cmp, jlt, add, etc. Instruction

Where are the local variables in memory?

The only way to store them is on a stack.

Why?

Function calls

LIFO

- Last in First Out
- Must need a “stack” like feature to implement them

Processor Stack

- Processors provide a stack pointer
- %esp on x86
  - Instructions like push and pop are provided by hardware
  - They automatically increment/decrement the stack pointer. On x86 stack grows downwards (substract from esp!)
  - Unlike a “stack data type” data structure, this “stack” is simply implemented with only the esp (as the “top”). The entire memory can be treated as the “array”.

Function calls

System stack, compilers, Languages

Compilers generate machine code with the ‘esp’. The pointer is initialized to a proper value at the time of fork-exec by the OS for each process.

Langauges like C which provide for function calls, and recursion also assume that they will run on processors with a stack support in hardware

Convention needed

How to use the stack for effective implementation of function calls ?

What goes on stack?

Local variables

Function Parameters

Return address of instruction which called a function !

Activation Record

Local Vars + parameters + return address

When functions call each other

One activation record is built on stack for each function call

On function return, the record is destroyed

On x86

ebp and esp pointers are used to denote the activation record.

How? We will see soon. You may start exploring with “gcc -S” output assembly code.

X86 instructions

leave

Equivalent to

```
mov   %ebp, %esp     # esp = ebp
pop   %ebp
```

ret

Equivalent to

```
pop %ecx
Jmp %ecx
```

call x

Equivalent to

```
push %eip
jmp  x
```

X86 instructions

endbr64

Normally a NOP

Let’s see some examples now

Let’s compile using

gcc -S

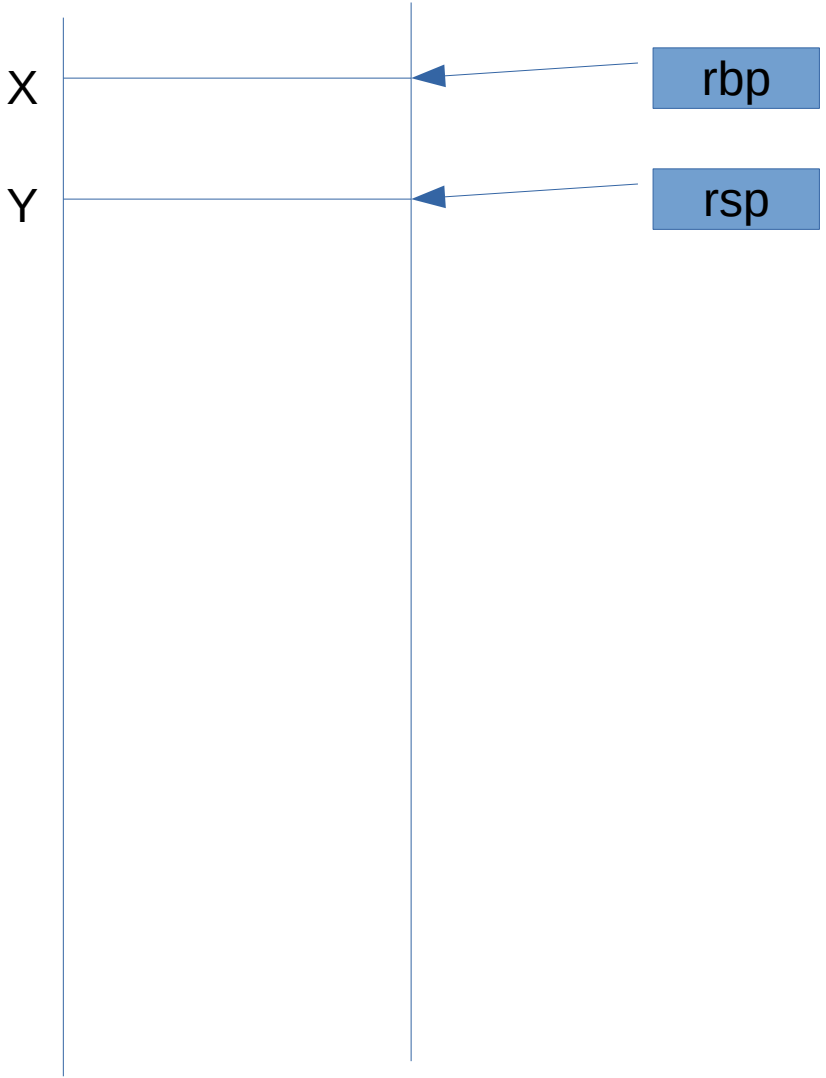
See code and understand

simple.c and simple.s

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```

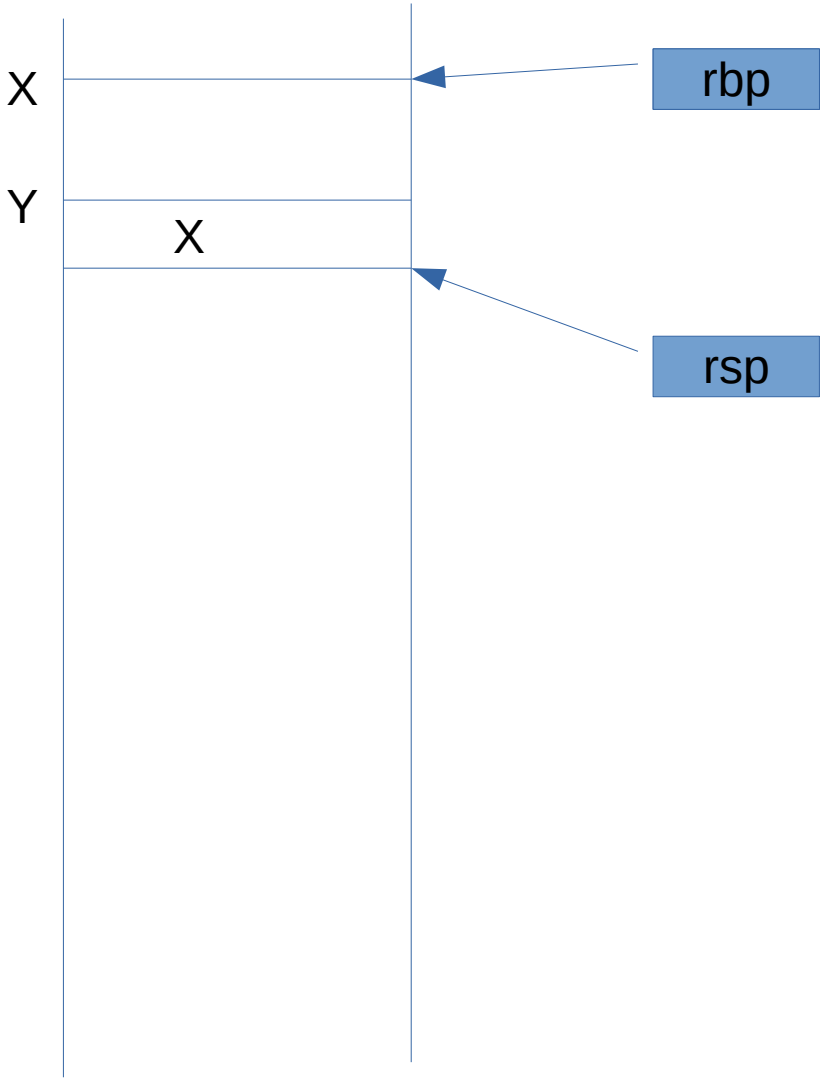
```
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $20, -8(%rbp)
    movl    $30, -4(%rbp)
    movl    -8(%rbp), %eax
    movl    %eax, %edi
    call    f
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    leave
    ret
```

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```



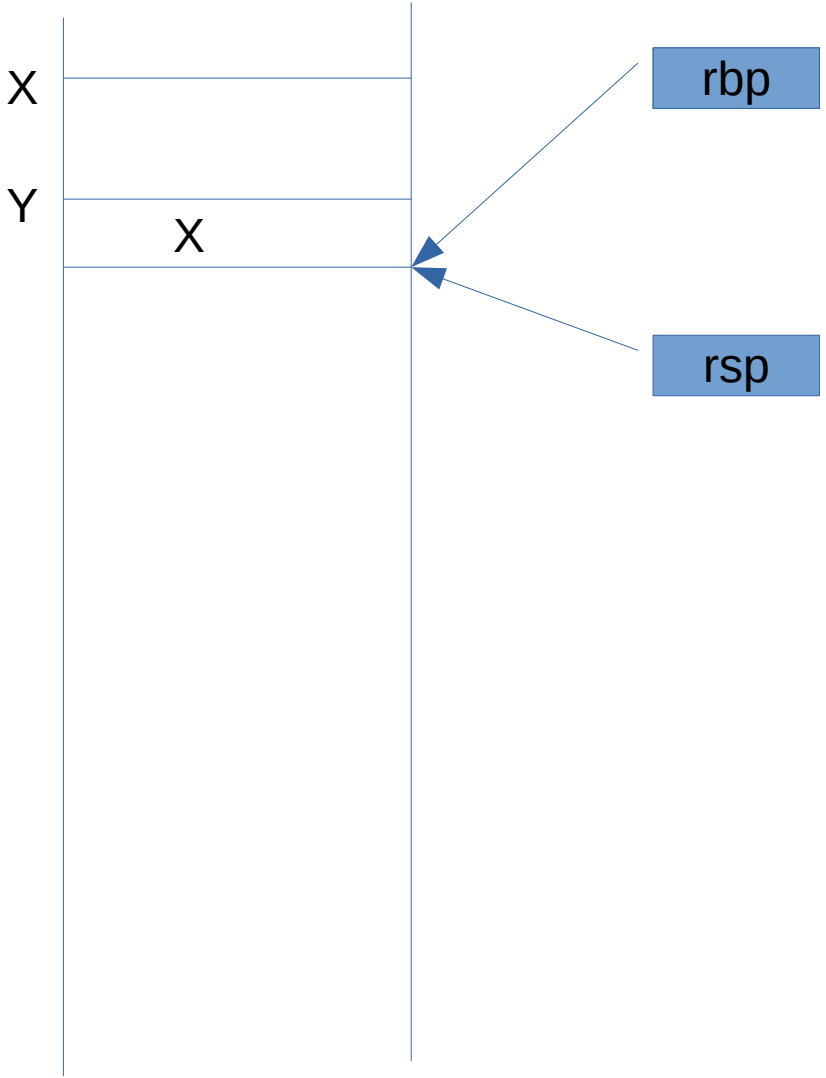

```
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $20, -8(%rbp)
    movl    $30, -4(%rbp)
    movl    -8(%rbp), %eax
    movl    %eax, %edi
    call    f
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    leave
    ret
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



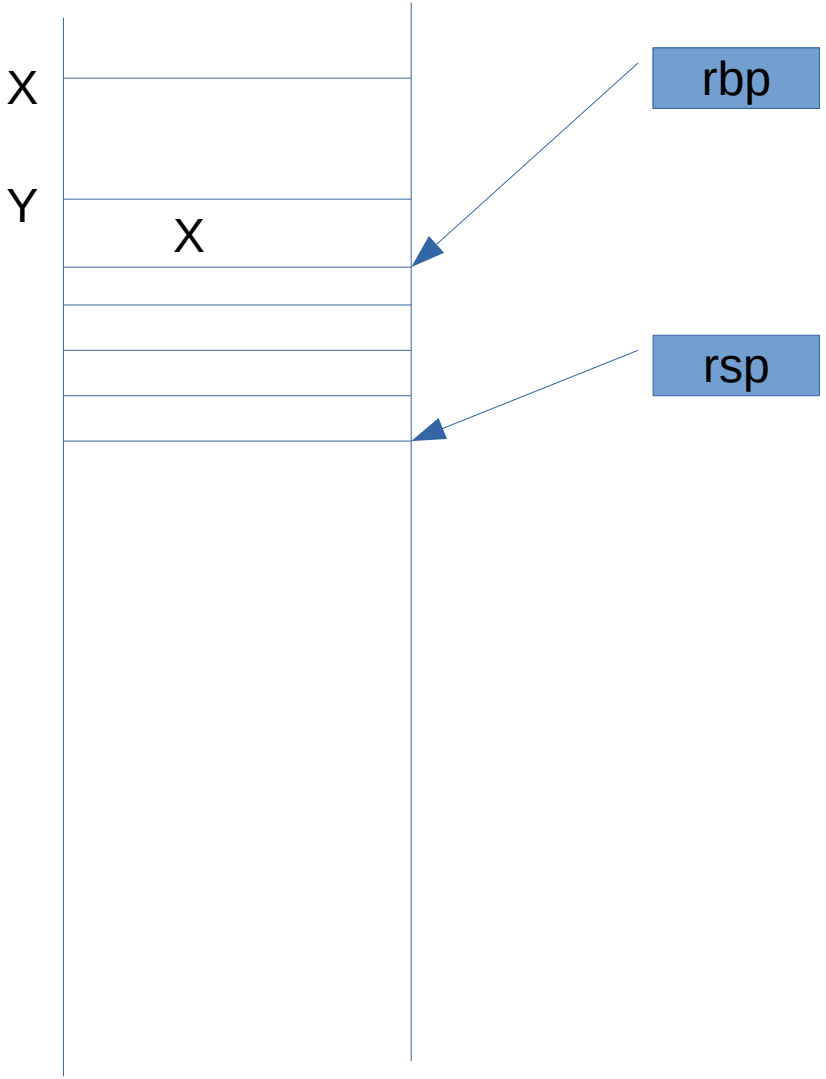

```
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $20, -8(%rbp)
    movl    $30, -4(%rbp)
    movl    -8(%rbp), %eax
    movl    %eax, %edi
    call    f
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    leave
    ret
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



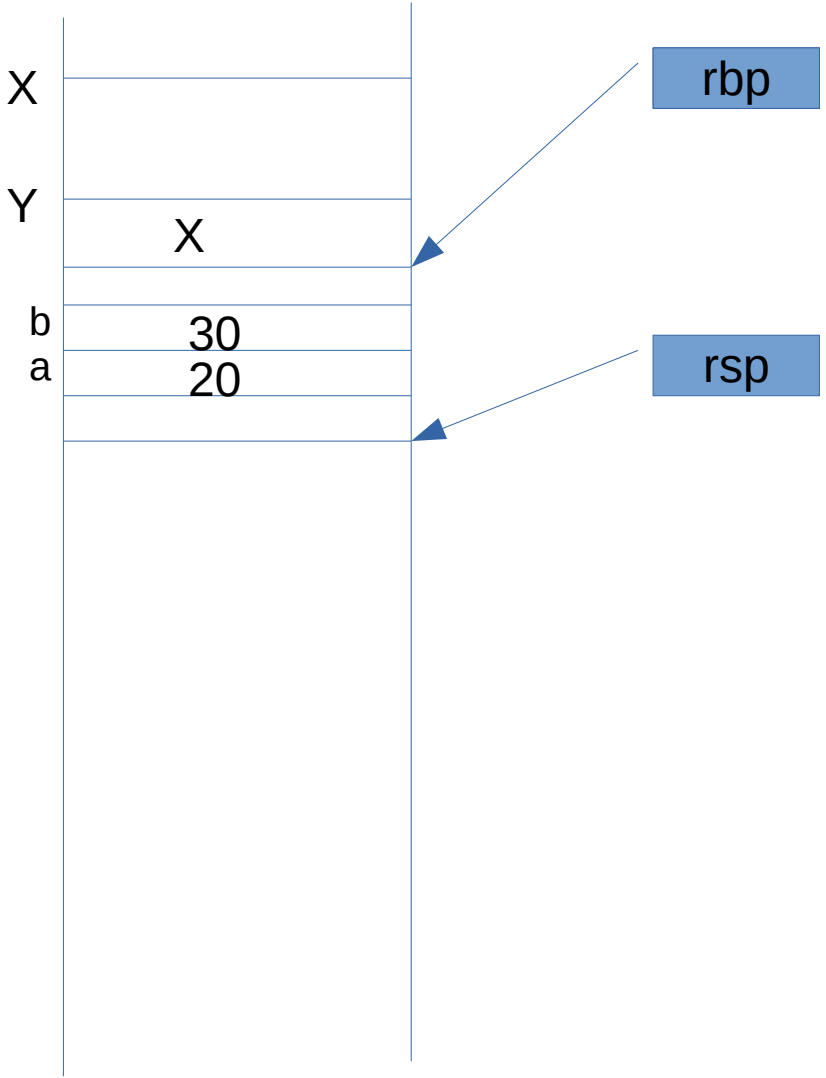

```
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $20, -8(%rbp)
    movl    $30, -4(%rbp)
    movl    -8(%rbp), %eax
    movl    %eax, %edi
    call    f
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    leave
    ret
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



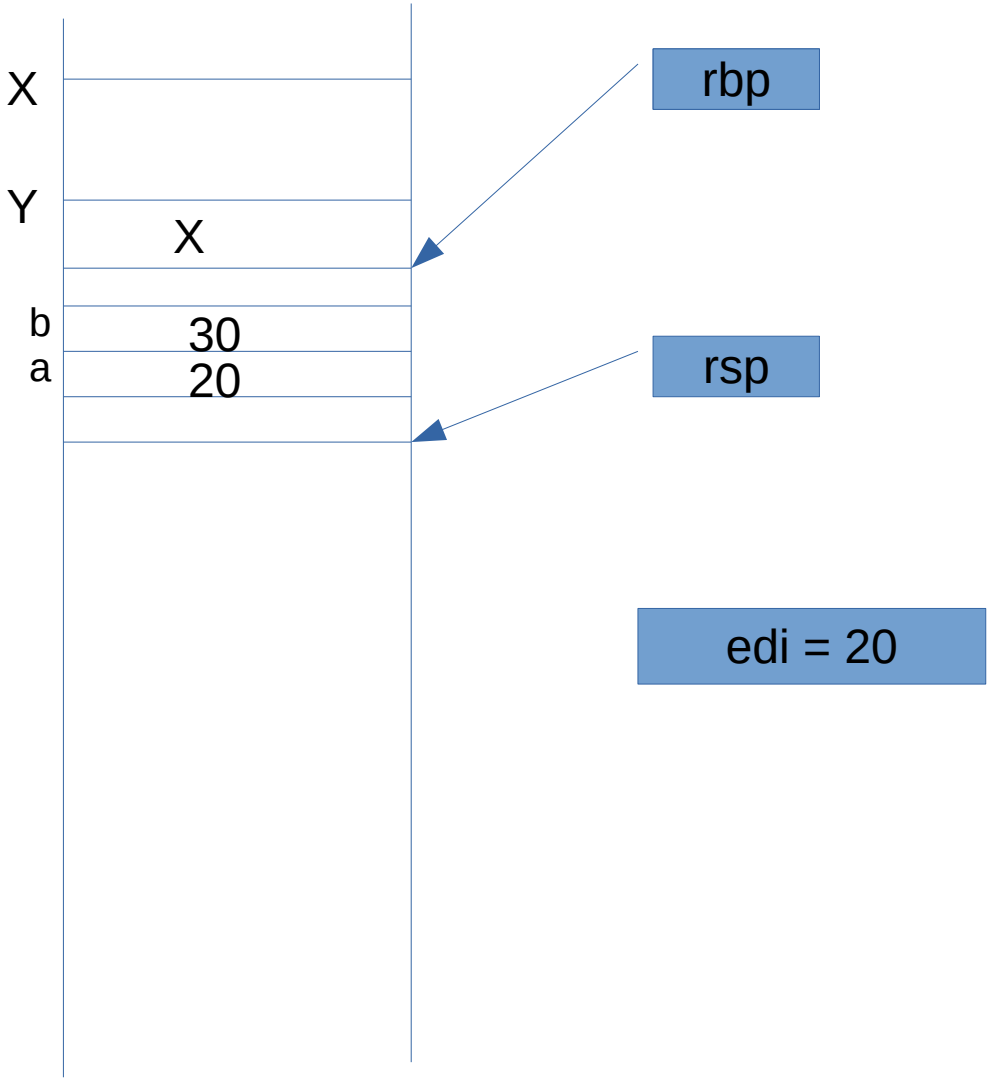

```
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $20, -8(%rbp)
    movl    $30, -4(%rbp)
    movl    -8(%rbp), %eax
    movl    %eax, %edi
    call    f
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    leave
    ret
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



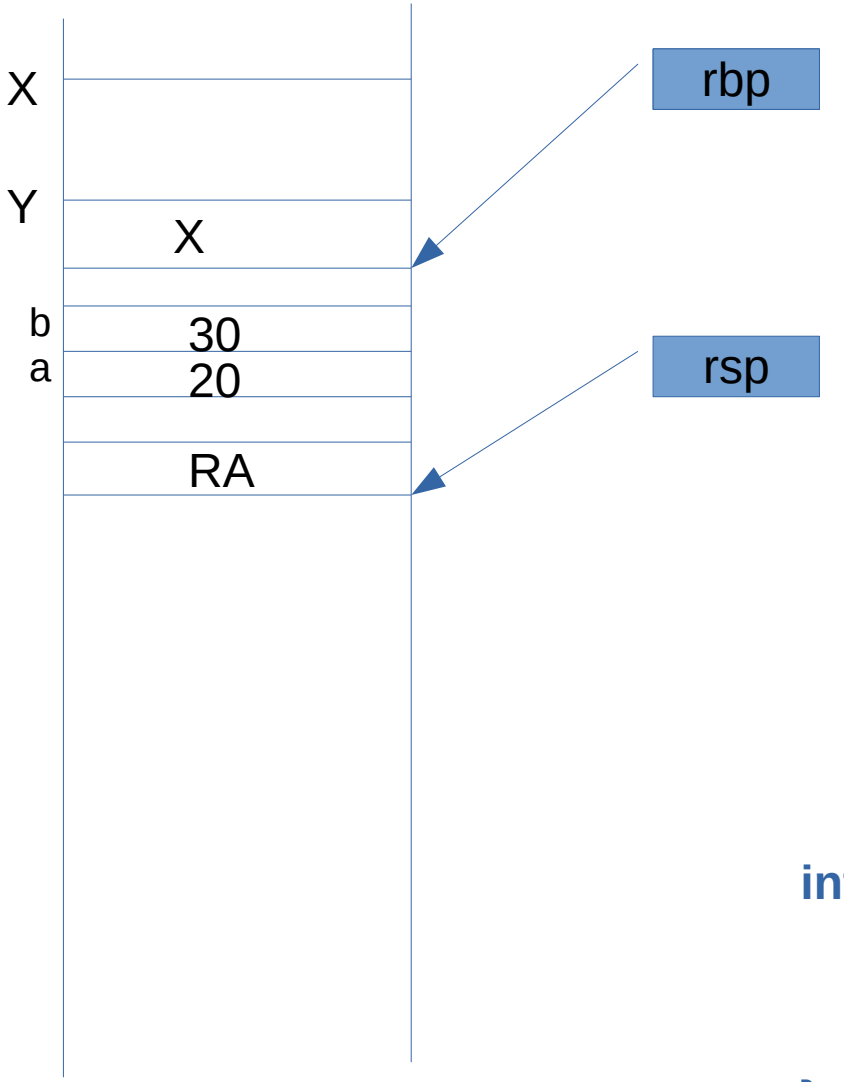

```
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $20, -8(%rbp)
    movl    $30, -4(%rbp)
    movl    -8(%rbp), %eax
    movl    %eax, %edi
    call    f
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    leave
    ret
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```




```
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $20, -8(%rbp)
    movl    $30, -4(%rbp)
    movl    -8(%rbp), %eax
    movl    %eax, %edi
    call    f
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    leave
    ret
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



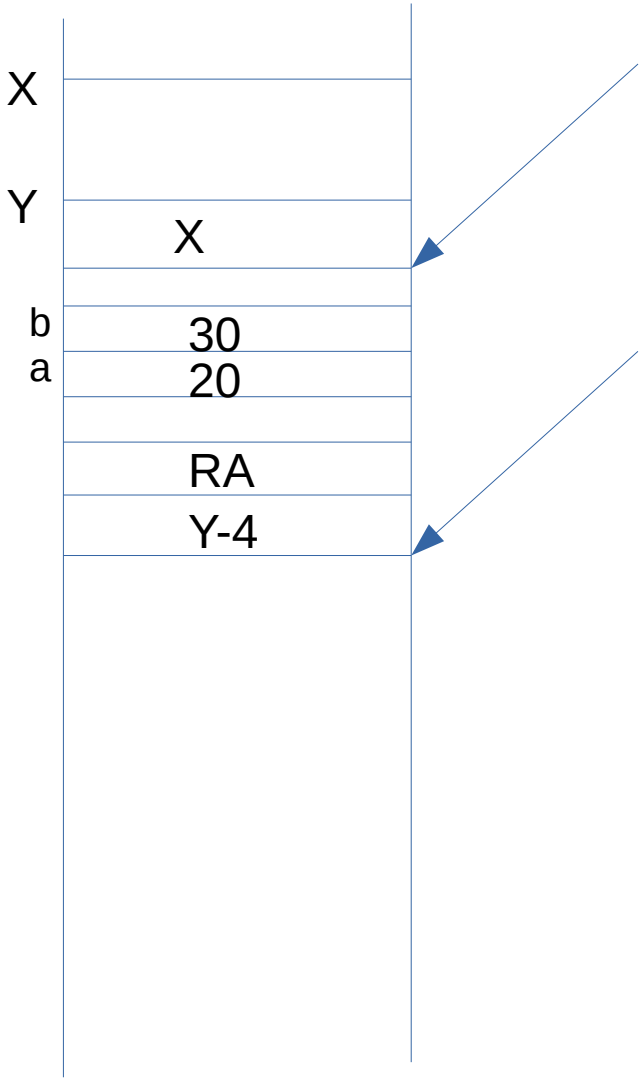

```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
Leave
ret
```

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

edi = 20

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



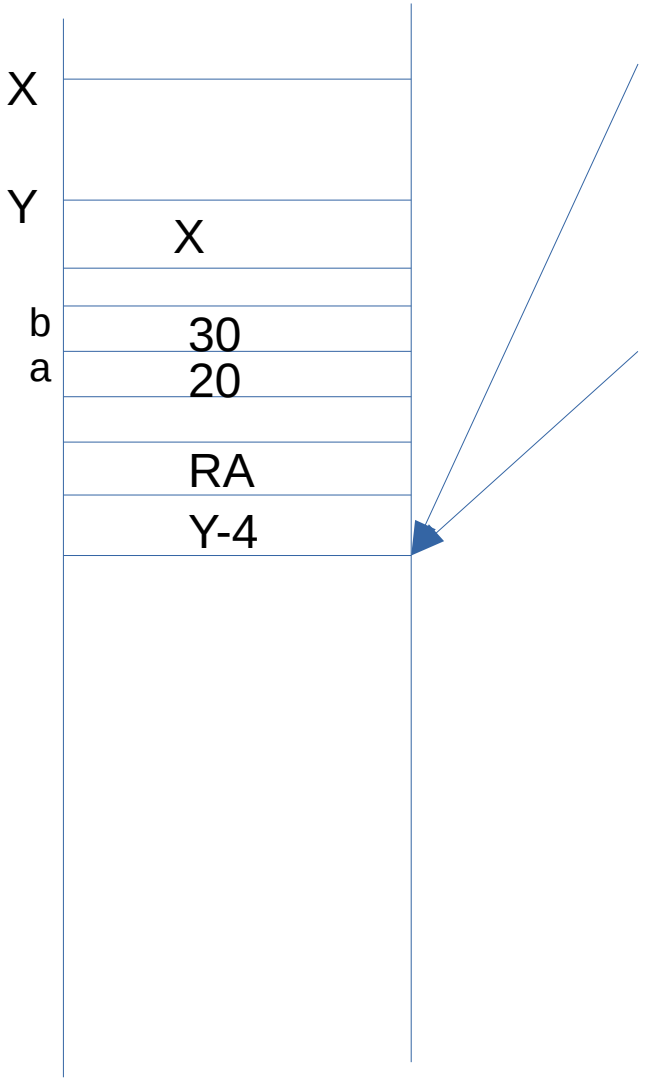

```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
Leave
ret
```

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

edi = 20

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



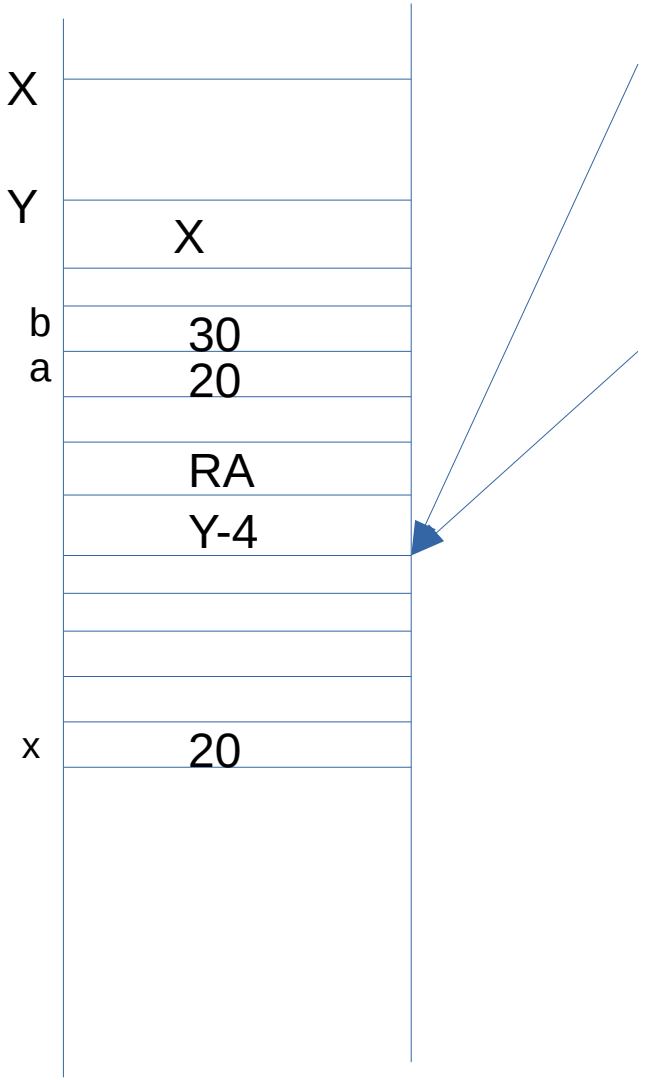

```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
Leave
ret
```

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

edi = 20

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



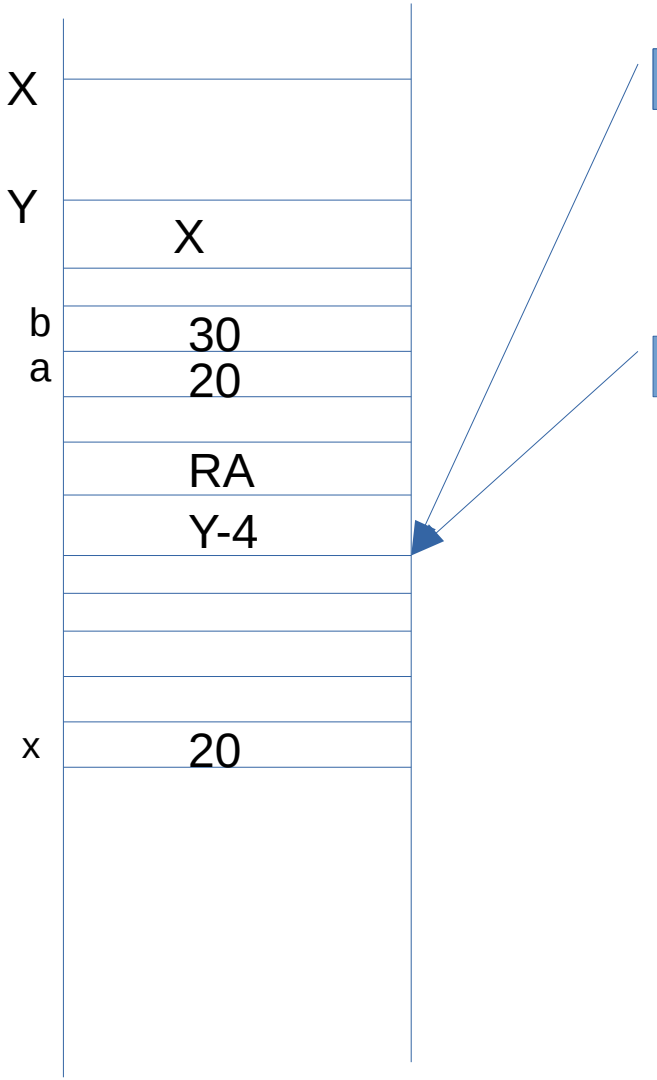

```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
Leave
ret
```

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

edi = 20

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```




```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
Leave
ret
```

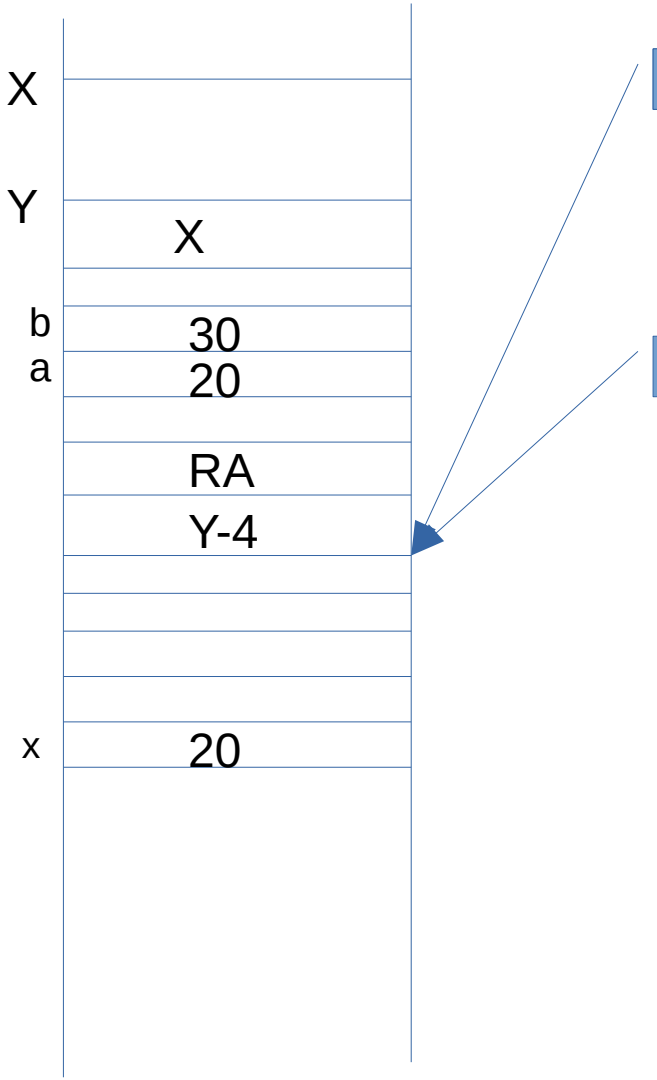
```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

edi = 20

eax = 20

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



rbp

rsp

```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
Leave
ret
```

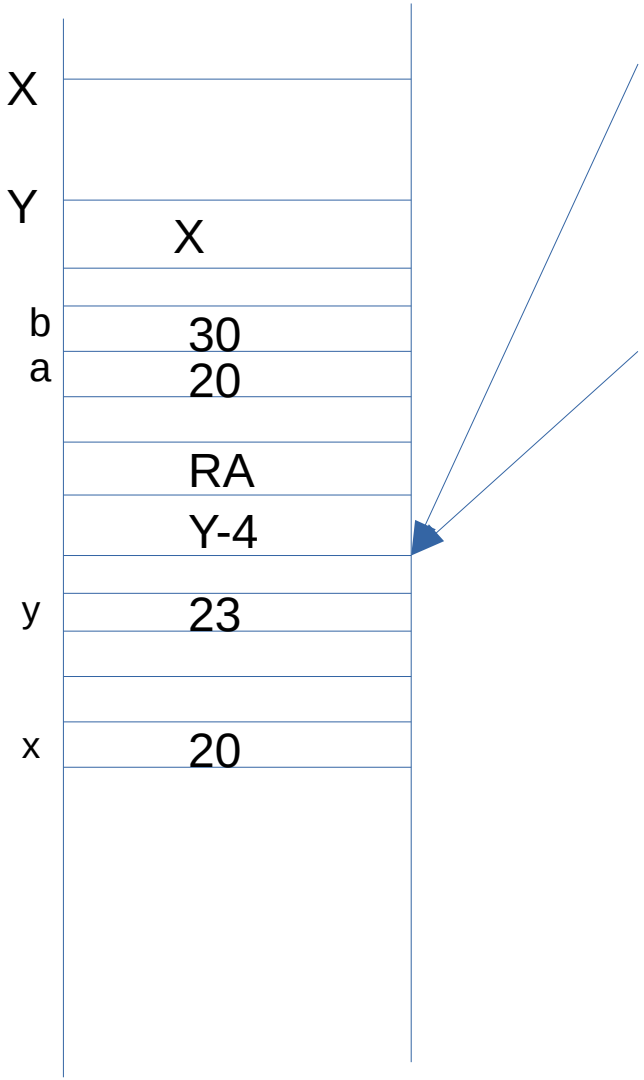
```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

edi = 20

eax = 23

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```




```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
Leave
ret
```

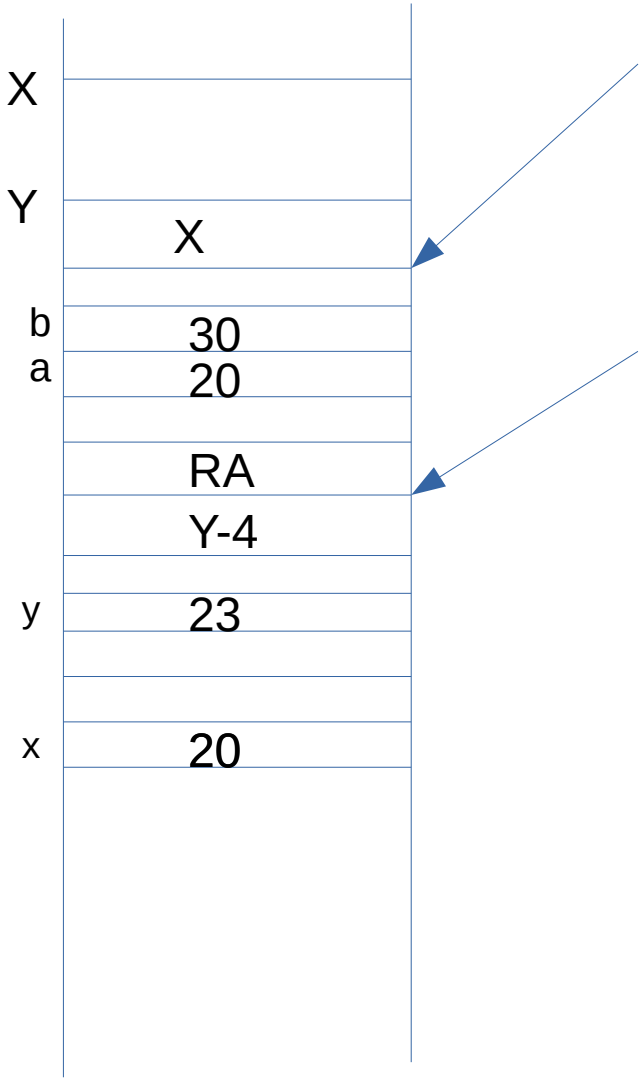
```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

edi = 20

eax = 23

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```




```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
Leave
ret
```

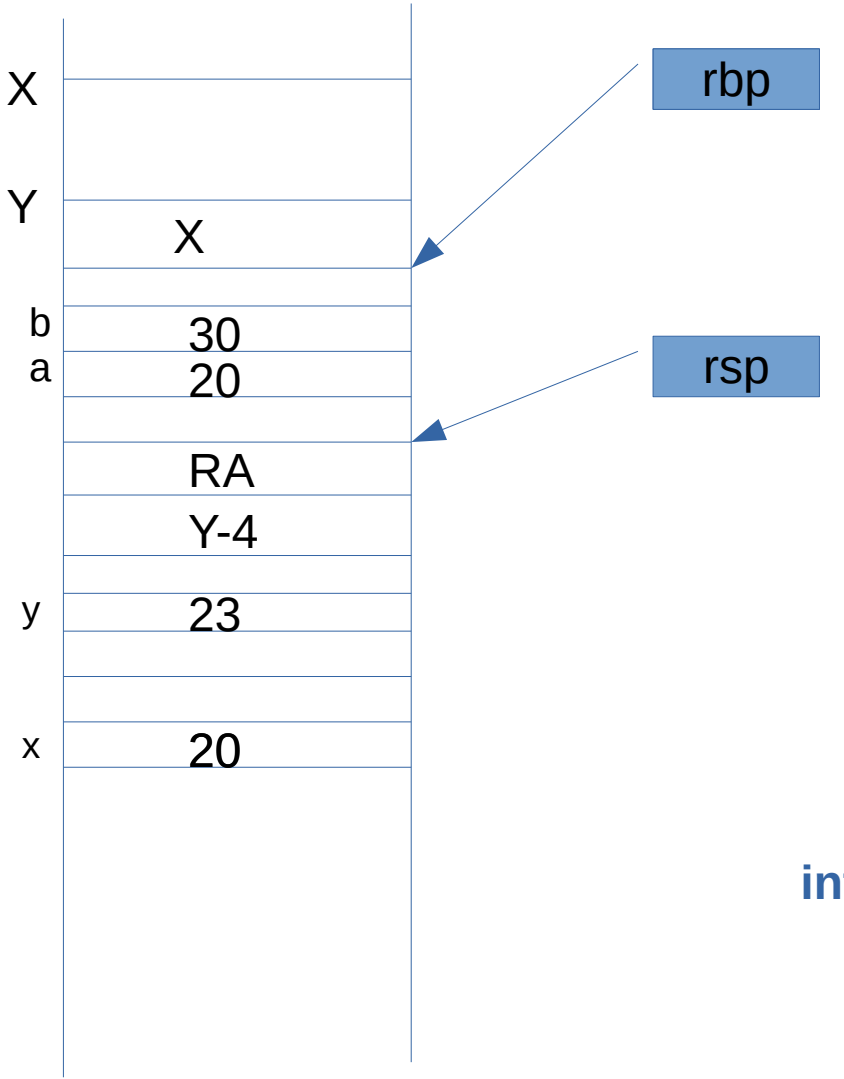
```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

edi = 20

Eax = 23

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```




```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
Leave
ret
```

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

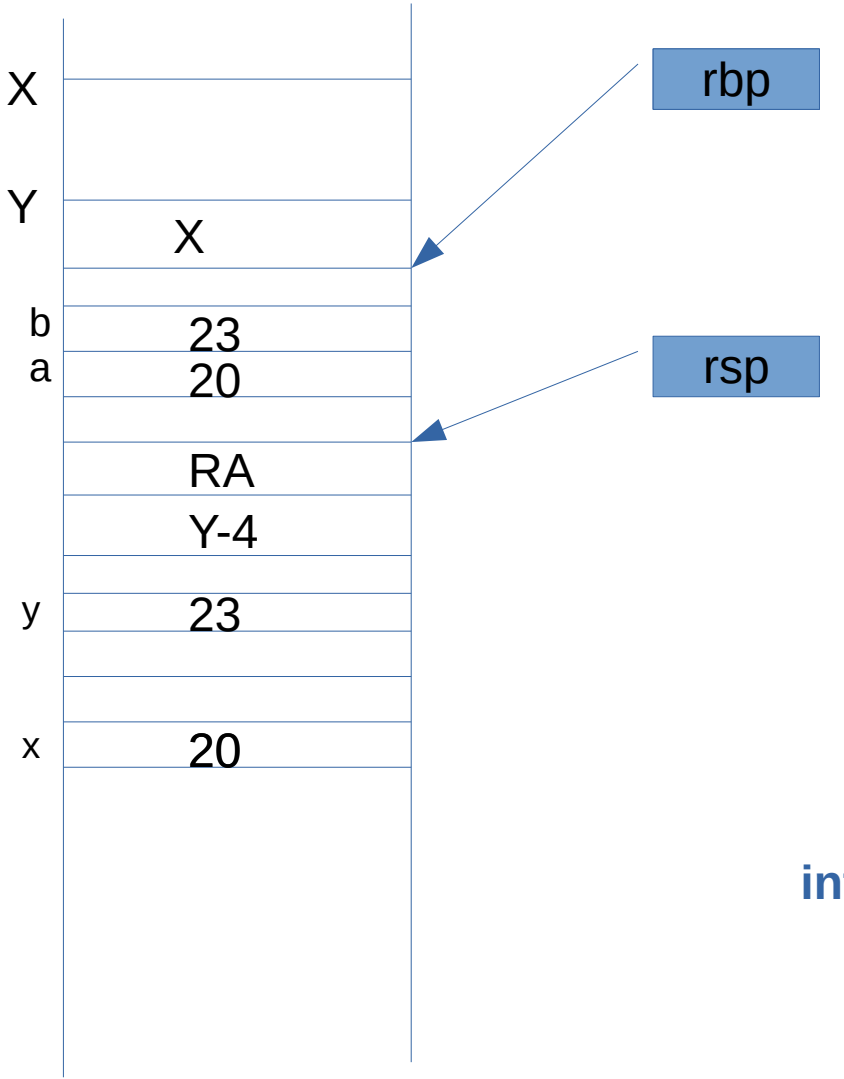
edi = 20

eax = 23

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```

eip = RA




```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
leave
ret
```

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

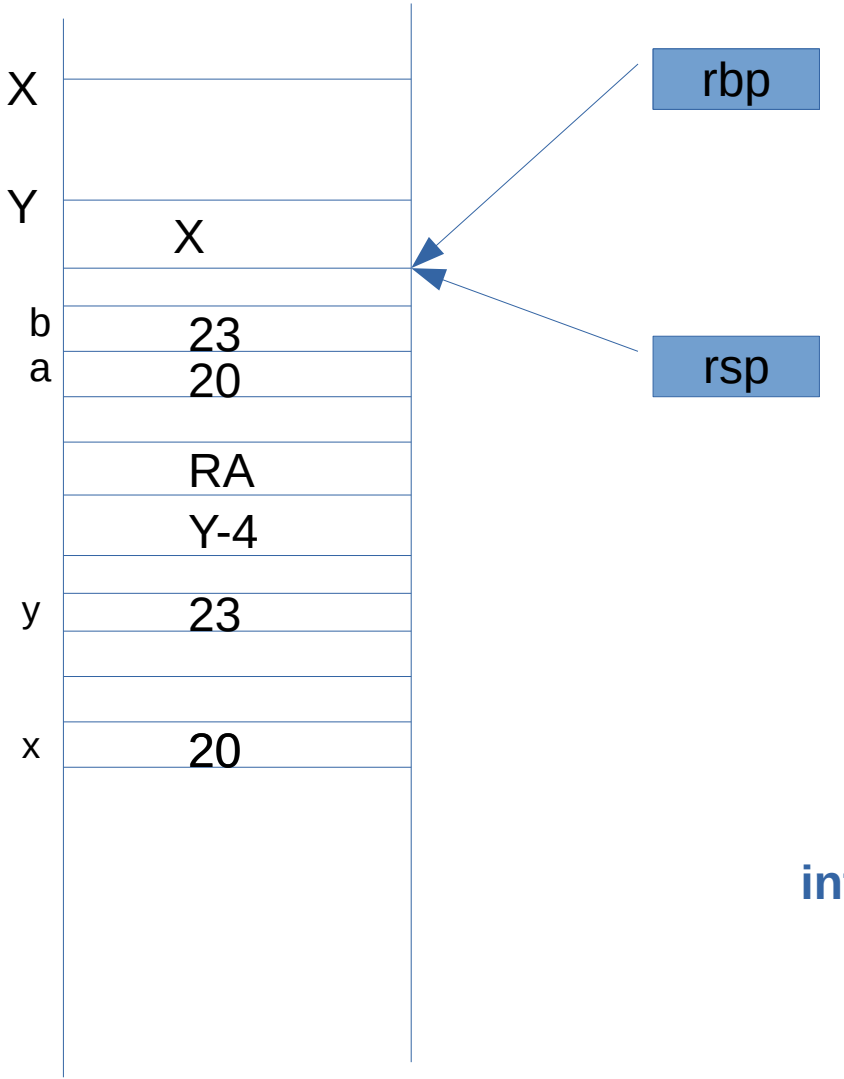
edi = 20

eax = 23

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```

eip = RA




```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
leave
# mov rbp rsp; pop rbp
ret
```

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

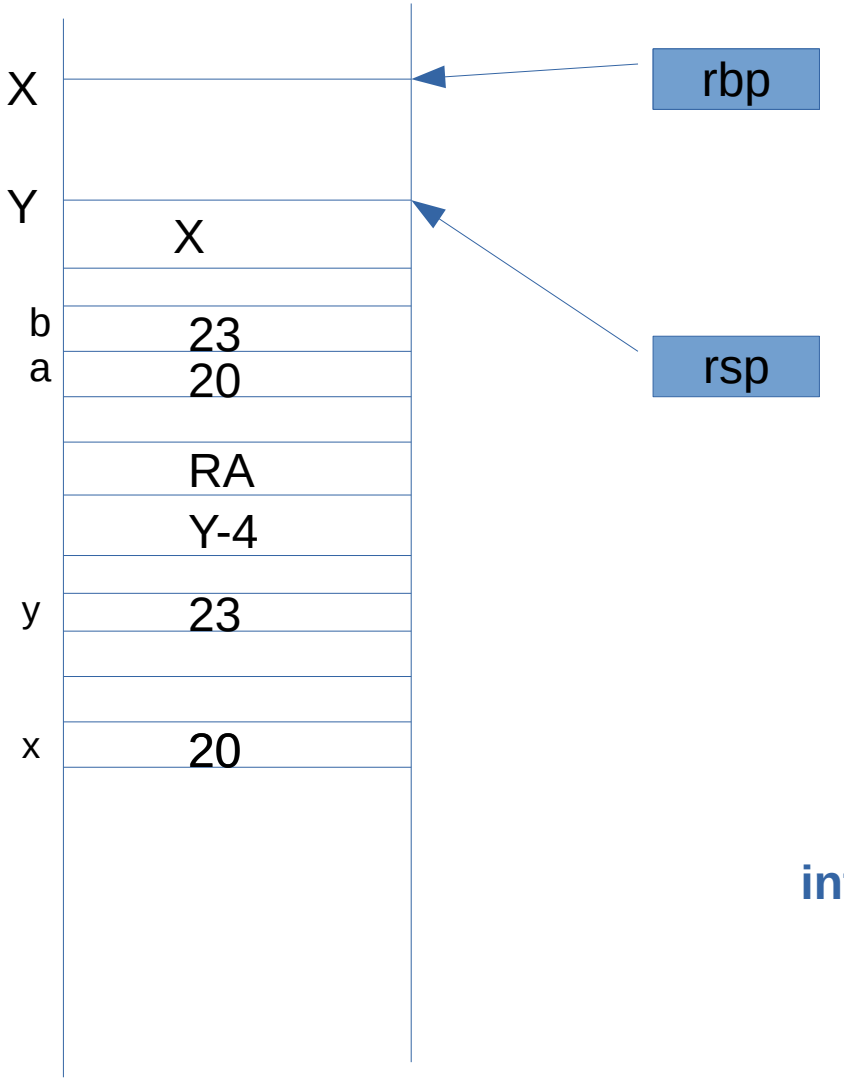
edi = 20

eax = 23

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```

eip = RA




```
main:
endbr64
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $20, -8(%rbp)
movl    $30, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %edi
call    f
RA:
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
leave
# mov ebp esp; pop ebp
ret
```

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    -20(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

edi = 20

eax = 23

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```

eip = RA

Further on calling convention

This was a simple program

The parameter was passed in a register!

What if there were many parameters?

CPUs have different numbers of registers.

More parameters, more functions demand a more sophisticated convention

May be slightly different on different processors, or 32-bit, 64-bit

Caller save and Callee save registers

Local variables

- Are visible only within the function
- Recursion: different copies of variables
- Stored on “stack”

Registers

- Are only one copy
- Are within the CPU

Local Variables & Registers conflict

- Compiler’s dillemma: While generating code for a function, which registers to use?
- The register might have been in use in earlier function call

Caller save and Callee save registers

Caller Save registers

Which registers need to be saved by caller function . They can be used by the callee function!

The caller function will push them (if already in use, otherwise no need) on the stack

Callee save registers

Will be pushed on to the stack by called (callee) function

How to return values?

On the stack itself – then caller will have to pop

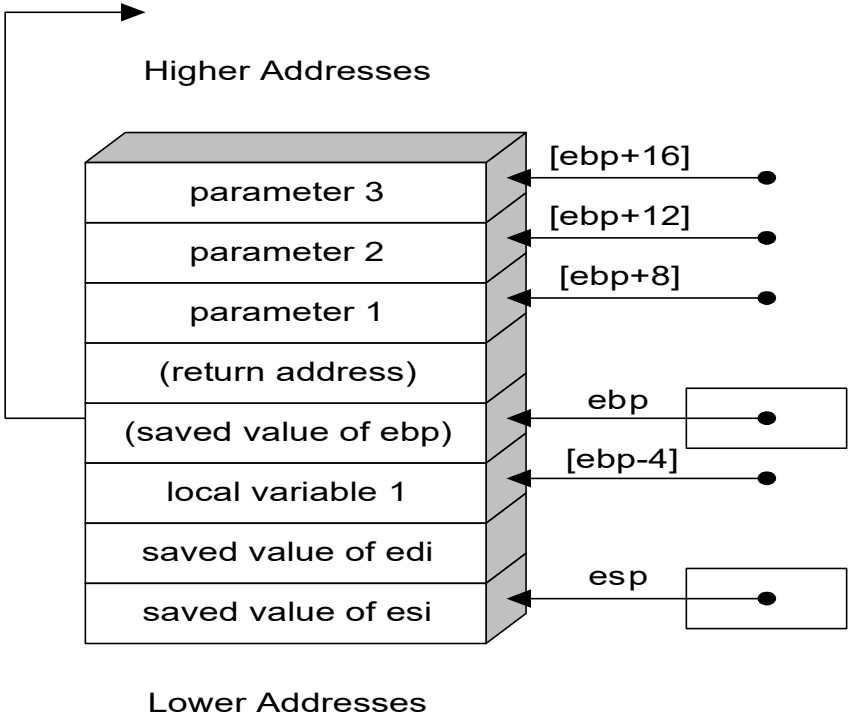
In a register, e.g. eax

X86 convention – caller, callee saved
32 bit

The caller-saved registers are EAX, ECX, EDX.

The callee-saved registers are EBX, EDI, and ESI

Activation record looks like this



F() called g()

Parameters-i refers to parameters passed by f() to g()

Local variable is a variable in g()

Return address is the location in f() where call should go back

X86 caller and callee rules(32 bit)

Caller rules on call

Push caller saved registers on stack

Push parameters on the stack – in reverse order. Why?

Substract esp, copy data

call f() // push + jmp

Caller rules on return

return value is in eax

remove parameters from stack : Add to esp.

Restore caller saved registers (if any)

X86 caller and callee rules

Callee rules on call

1) push ebp

   mov ebp, esp

ebp(+-offset) normally used to locate local vars and parameters on stack

ebp holds a copy of esp

Ebp is pushed so that it can be later popped while returnig

2) Allocate local variables

3) Save callee-saved registers

https://www.cs.virginia.edu/~evans/cs216/guides/x86.html

X86 caller and callee rules

Callee rules on return

1) Leave return value in eax
2) Restore callee saved registers
3) Deallocate local variables
4) restore the ebp
5) return

https://www.cs.virginia.edu/~evans/cs216/guides/x86.html

32 bit vs 64 bit calling convention

Registers are used for passing parameters in 64 bit , to a large extent

Upto 6 parameters

More parameters pushed on stack

See

https://aaronbloomfield.github.io/pdr/book/x86-64bit-ccc-chapter.pdf

Beware

When you read assembly code generated using

gcc -S

You will find

More complex instructions

But they will essentially follow the convention mentioned

Comparison

	MIPS	x86
Arguments:	First 4 in `%a0`–`%a3`, remainder on stack	Generally all on stack
Return values:	`%v0`–`%v1`	`%eax`
Caller-saved registers:	`%t0`–`%t9`	`%eax`, `%ecx`, & `%edx`
Callee-saved registers:	`%s0`–`%s9`	Usually none

Figure 6.2: A comparison of the calling conventions of MIPS and x86

From the textbook by Misruda

simple3.c and simple3.s

```
int f(int x1, int x2, int x3, int x4, int x5, int x6, int x7, int x8, int x9, int x10) {
    int h;
    h = x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10 + 3;
    return h;
}
int main() {
    int a1 = 10, a2 = 20, a3 = 30, a4 = 40, a5 = 50, a6 = 60, a7 = 70, a8 = 80, a9 = 90, a10 = 100;
    int b;
    b = f(a1, a2, a3, a4, a5, a6, a7, a8, a9, a10);
    return b;
}
```

simple3.c and simple3.s

```
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $48, %rsp
    movl    $10, -44(%rbp)
    movl    $20, -40(%rbp)
    movl    $30, -36(%rbp)
    movl    $40, -32(%rbp)
    movl    $50, -28(%rbp)
    movl    $60, -24(%rbp)
    movl    $70, -20(%rbp)
    movl    $80, -16(%rbp)
    movl    $90, -12(%rbp)
    movl    $100, -8(%rbp)
    movl    -24(%rbp), %r9d
    movl    -28(%rbp), %r8d
    movl    -32(%rbp), %ecx
    movl    -36(%rbp), %edx
    movl    -40(%rbp), %esi
    movl    -44(%rbp), %eax
    movl    -8(%rbp), %edi
    pushq   %rdi
    movl    -12(%rbp), %edi
    pushq   %rdi
    movl    -16(%rbp), %edi
    pushq   %rdi
    movl    -20(%rbp), %edi
    pushq   %rdi
    movl    %eax, %edi
    call    f
    addq    $32, %rsp
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    leave
    ret
```

simple3.c and simple3.s

```
f:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
    movl    %edx, -28(%rbp)
    movl    %ecx, -32(%rbp)
    movl    %r8d, -36(%rbp)
    movl    %r9d, -40(%rbp)
    movl    -20(%rbp), %edx
    movl    -24(%rbp), %eax
    addl    %eax, %edx
    movl    -28(%rbp), %eax
    addl    %eax, %edx
    movl    -32(%rbp), %eax
    addl    %eax, %edx
    movl    -36(%rbp), %eax
    addl    %eax, %edx
    movl    -40(%rbp), %eax
    addl    %eax, %edx
    movl    16(%rbp), %eax
    addl    %eax, %edx
    movl    24(%rbp), %eax
    addl    %eax, %edx
    movl    32(%rbp), %eax
    addl    %eax, %edx
    movl    40(%rbp), %eax
    addl    %edx, %eax
    addl    $3, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

Let's see a demo of how the stack is built and destroyed during function calls, on a Linux machine using GCC.

Consider this C code

```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e);
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

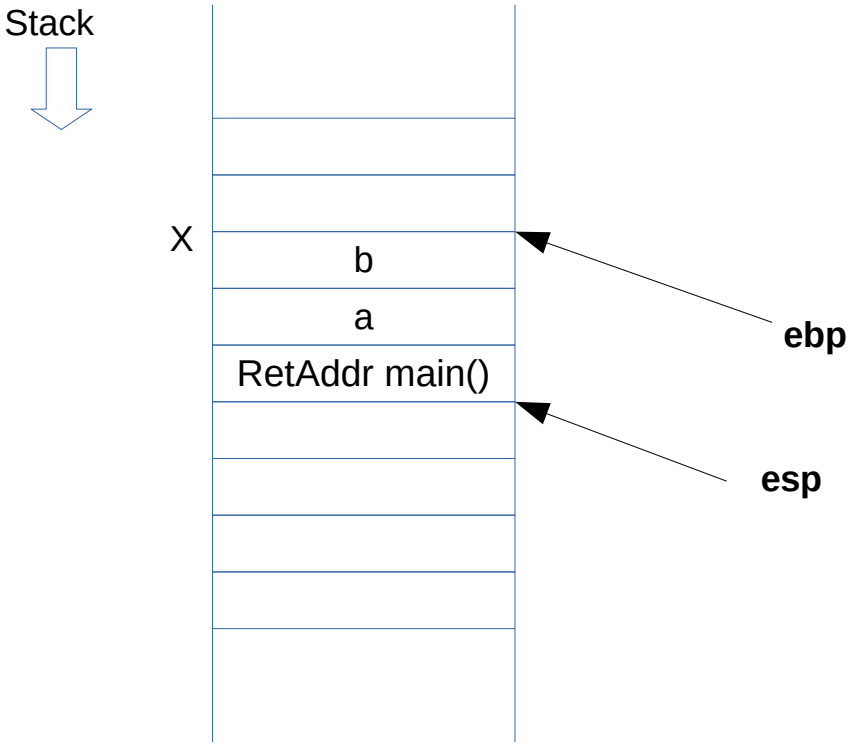
Translated to assembly as:

add:

```
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    8(%ebp), %edx
        movl    12(%ebp), %eax
        addl    %edx, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        leave
        ret
```

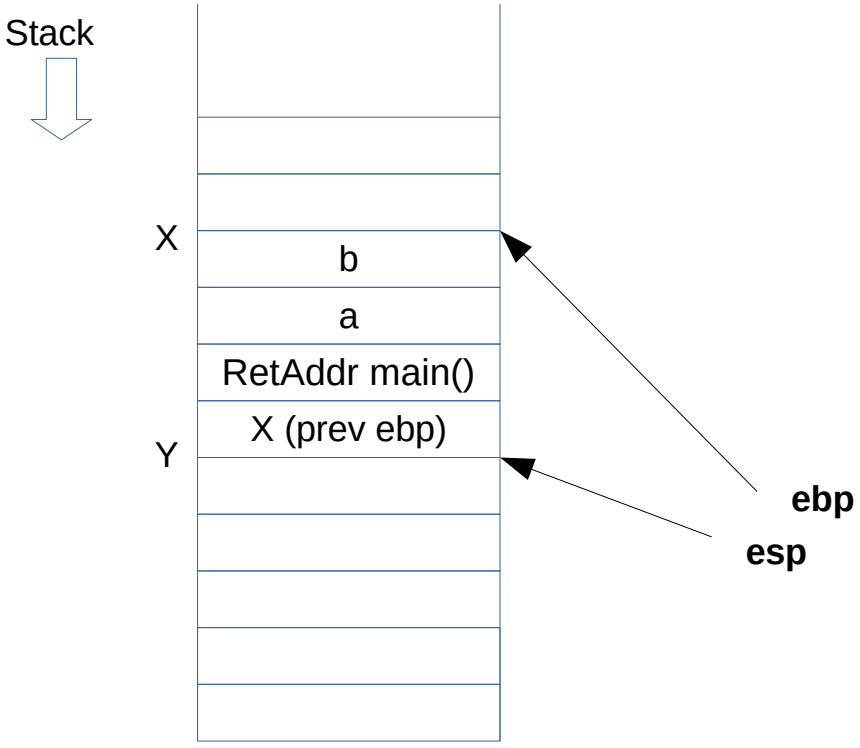
mult:

```
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $20, -24(%ebp)
        movl    $30, -20(%ebp)
        subl    $8, %esp
        pushl   -20(%ebp)
        pushl   -24(%ebp)
        call    add
        addl    $16, %esp
        movl    %eax, -16(%ebp)
        movl    8(%ebp), %eax
        imull   12(%ebp), %eax
        movl    %eax, %edx
        movl    -16(%ebp), %eax
        addl    %edx, %eax
        movl    %eax, -12(%ebp)
        movl    -12(%ebp), %eax
        leave
        ret
```



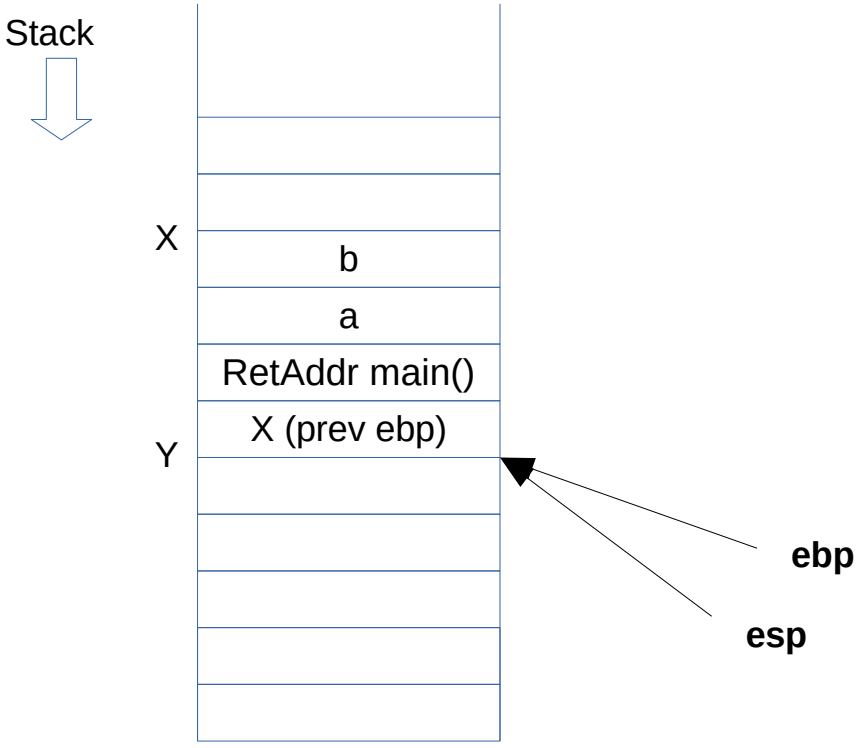

```
/* Control is here */
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e);
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

```
* Control is here */
mult:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $20, -24(%ebp)
        movl    $30, -20(%ebp)
        subl    $8, %esp
        pushl   -20(%ebp)
        pushl   -24(%ebp)
        call    add
```



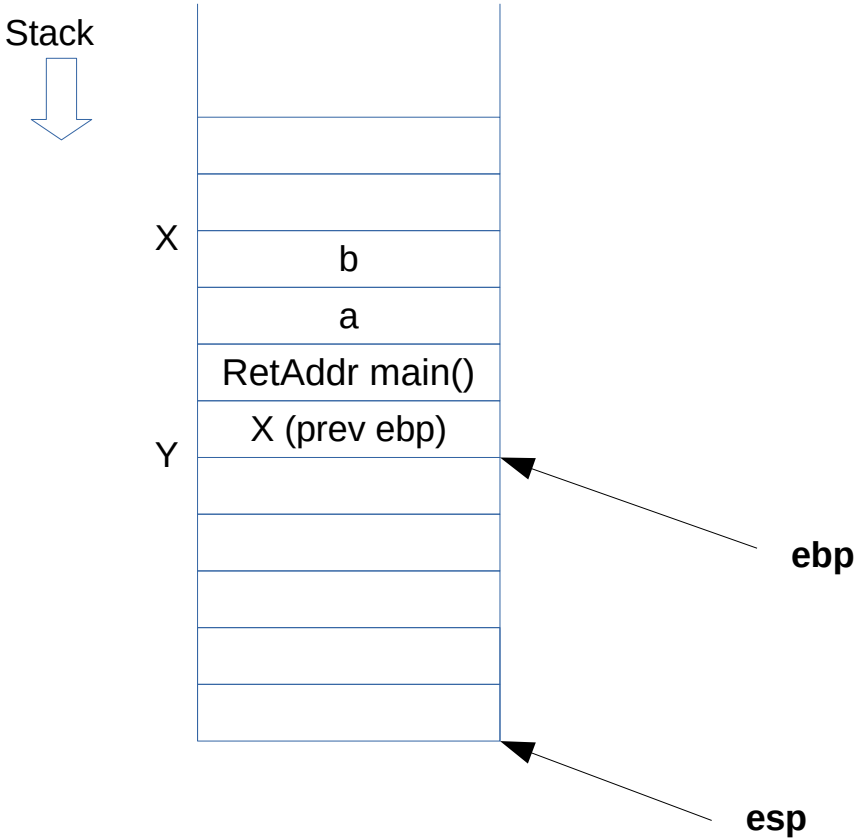

```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e);
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

```
mult:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $20, -24(%ebp)
        movl    $30, -20(%ebp)
        subl    $8, %esp
        pushl   -20(%ebp)
        pushl   -24(%ebp)
        call    add
```



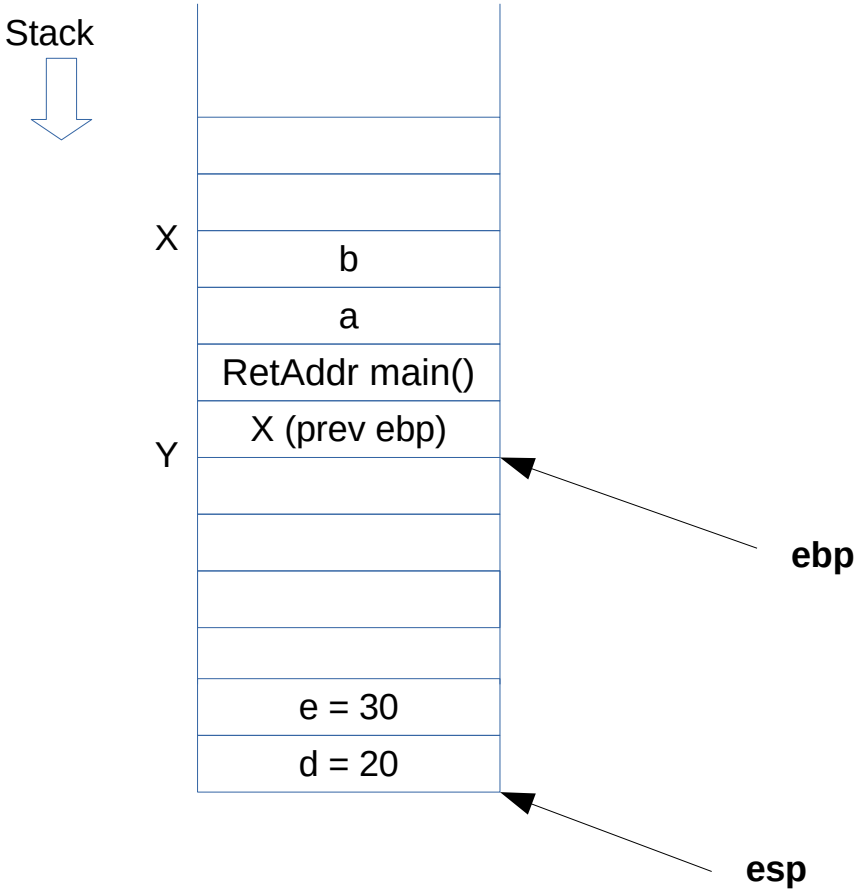

```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e);
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

```
mult:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $20, -24(%ebp)
        movl    $30, -20(%ebp)
        subl    $8, %esp
        pushl   -20(%ebp)
        pushl   -24(%ebp)
        call    add
```



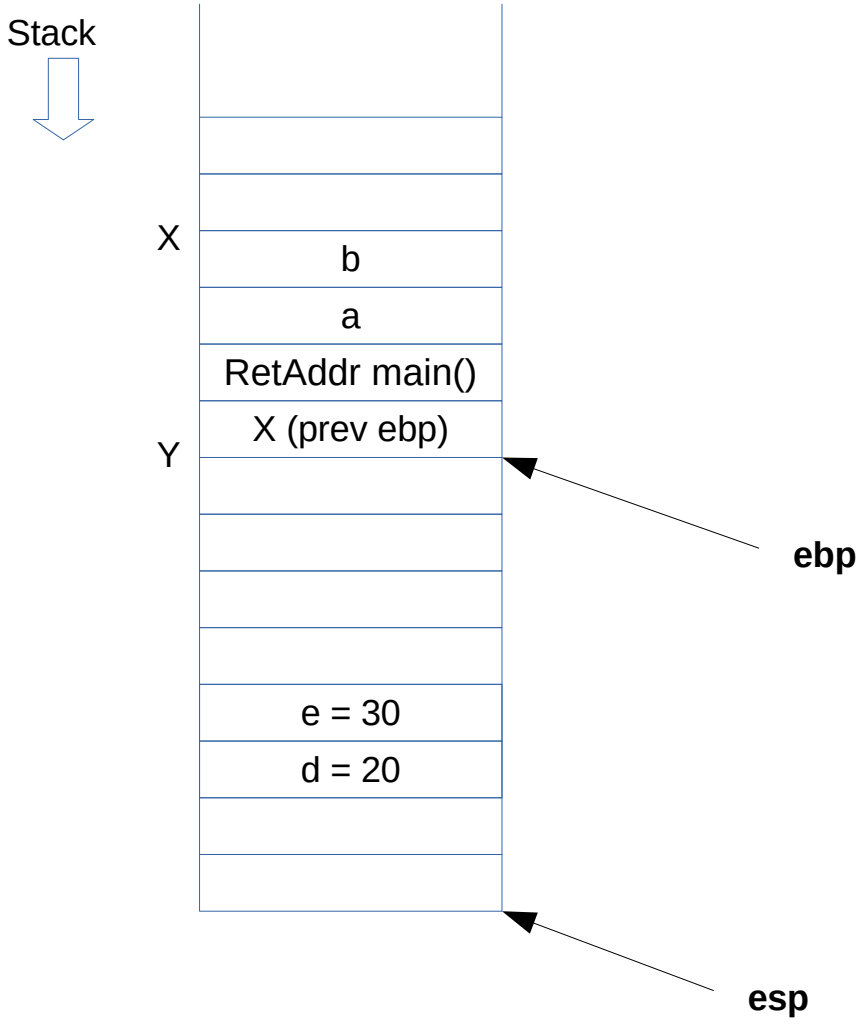

```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e);
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

```
mult:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $20, -24(%ebp)
        movl    $30, -20(%ebp)
        subl    $8, %esp
        pushl   -20(%ebp)
        pushl   -24(%ebp)
        call    add
```



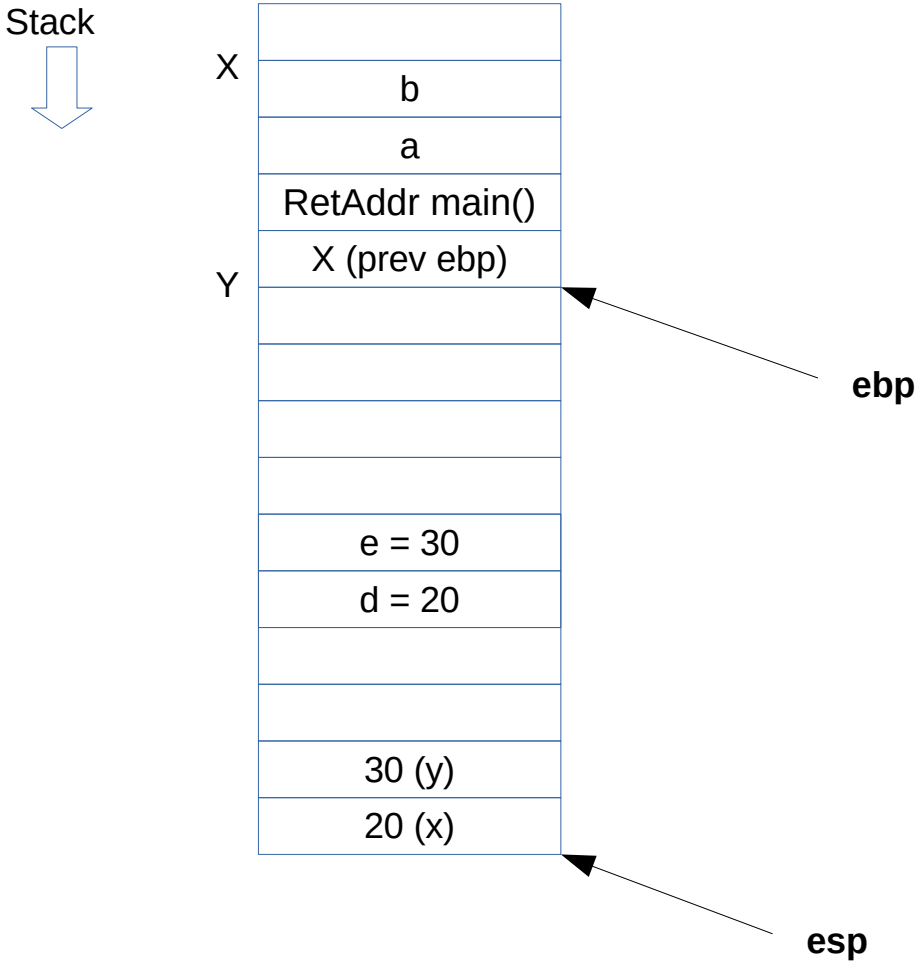

```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e);
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

```
mult:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $20, -24(%ebp)
        movl    $30, -20(%ebp)
        subl    $8, %esp
        pushl   -20(%ebp)
        pushl   -24(%ebp)
        call    add
```



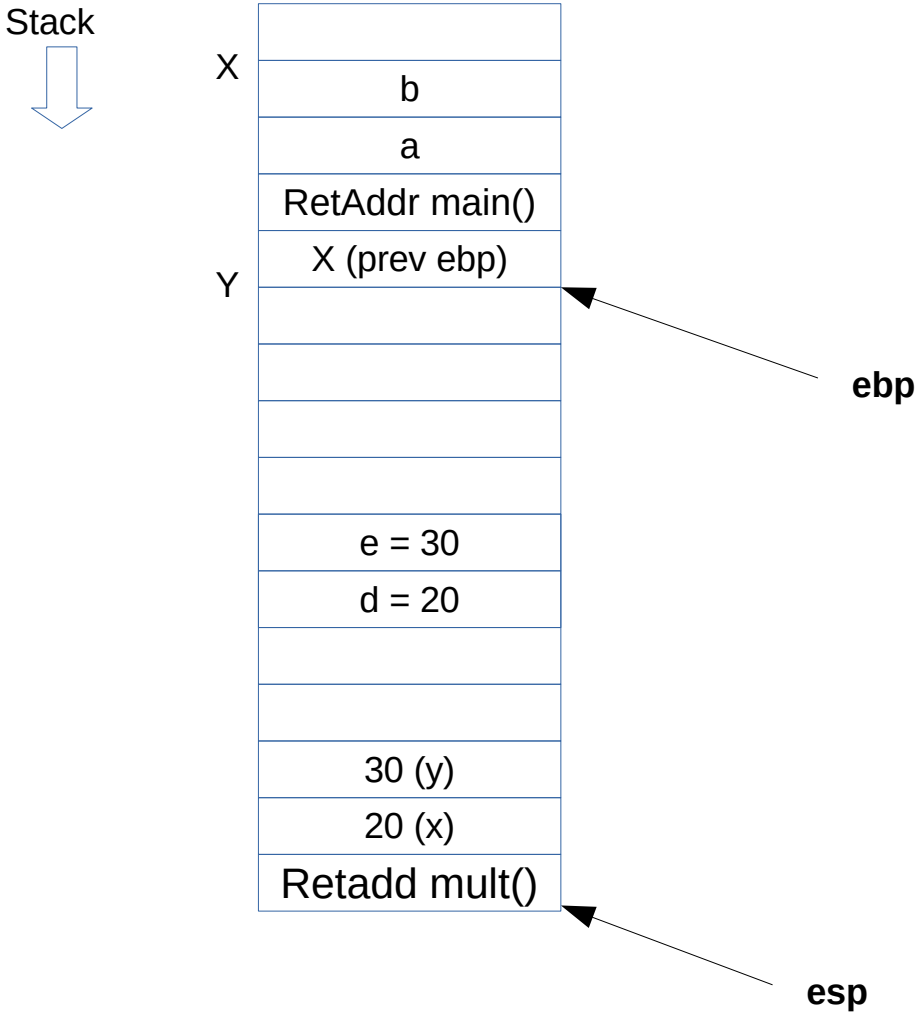

```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e);
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

```
mult:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $20, -24(%ebp)
        movl    $30, -20(%ebp)
        subl    $8, %esp
        pushl   -20(%ebp)
        pushl   -24(%ebp)
        call    add
```



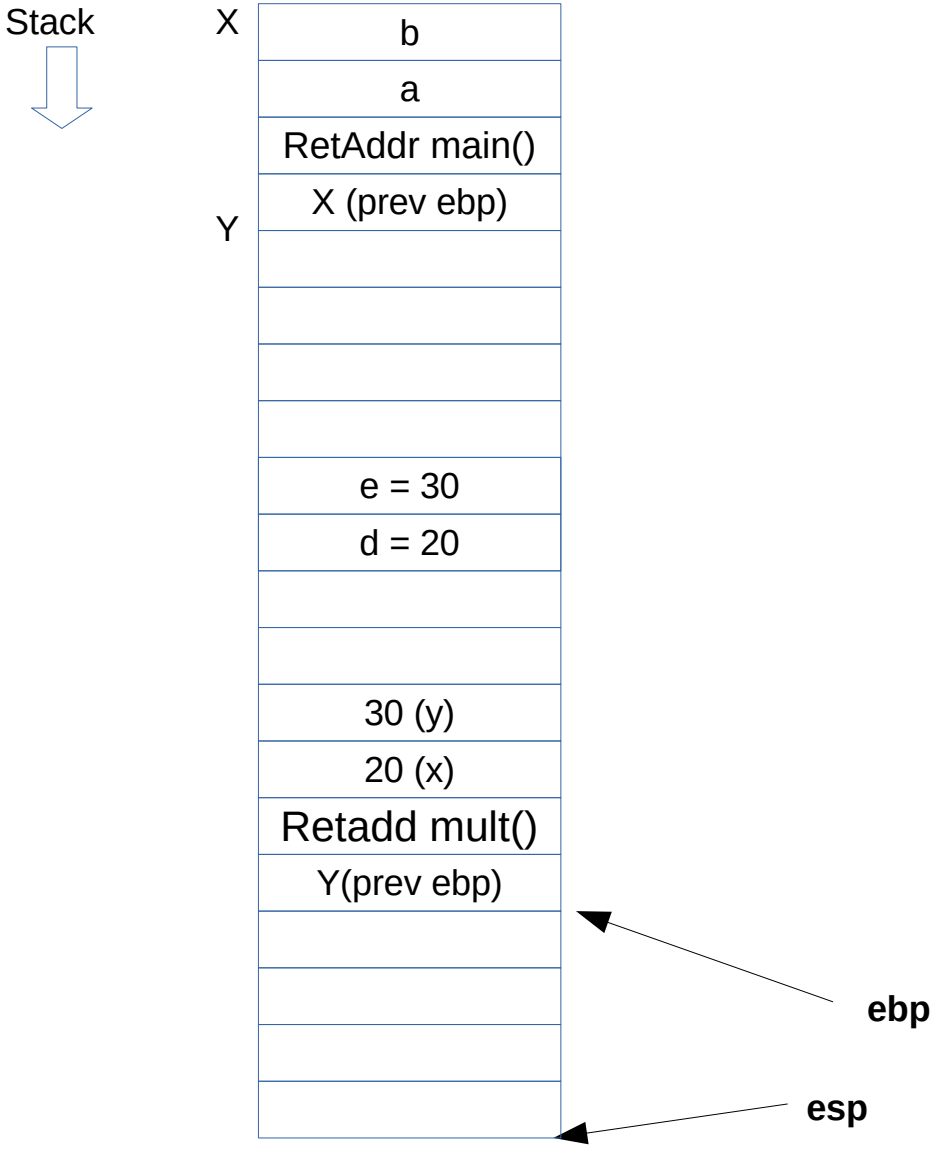

```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e);
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

```
mult:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $20, -24(%ebp)
        movl    $30, -20(%ebp)
        subl    $8, %esp
        pushl   -20(%ebp)
        pushl   -24(%ebp)
        call    add
```



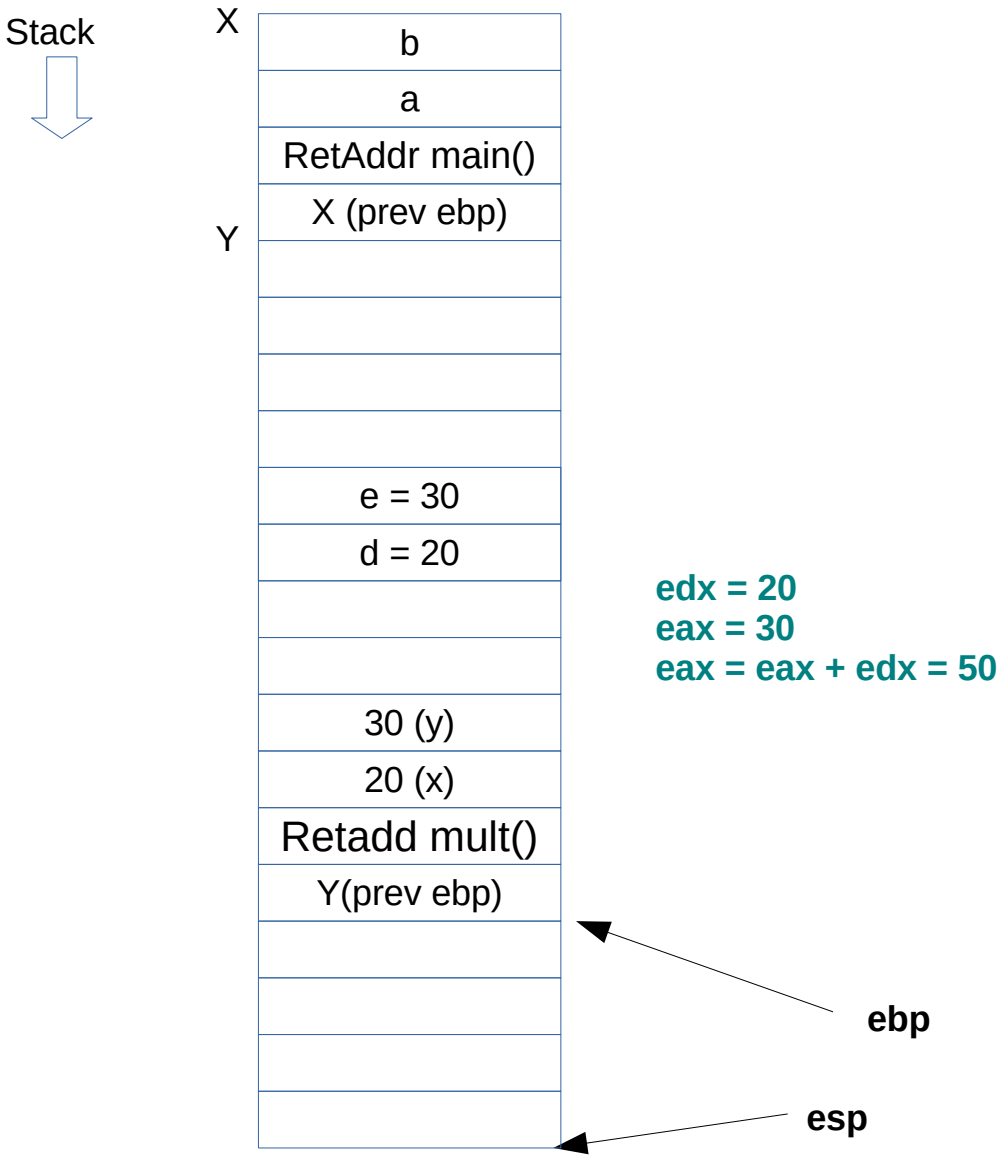

```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e);
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

```
mult:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $20, -24(%ebp)
        movl    $30, -20(%ebp)
        subl    $8, %esp
        pushl   -20(%ebp)
        pushl   -24(%ebp)
        call    add
```




```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e);
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

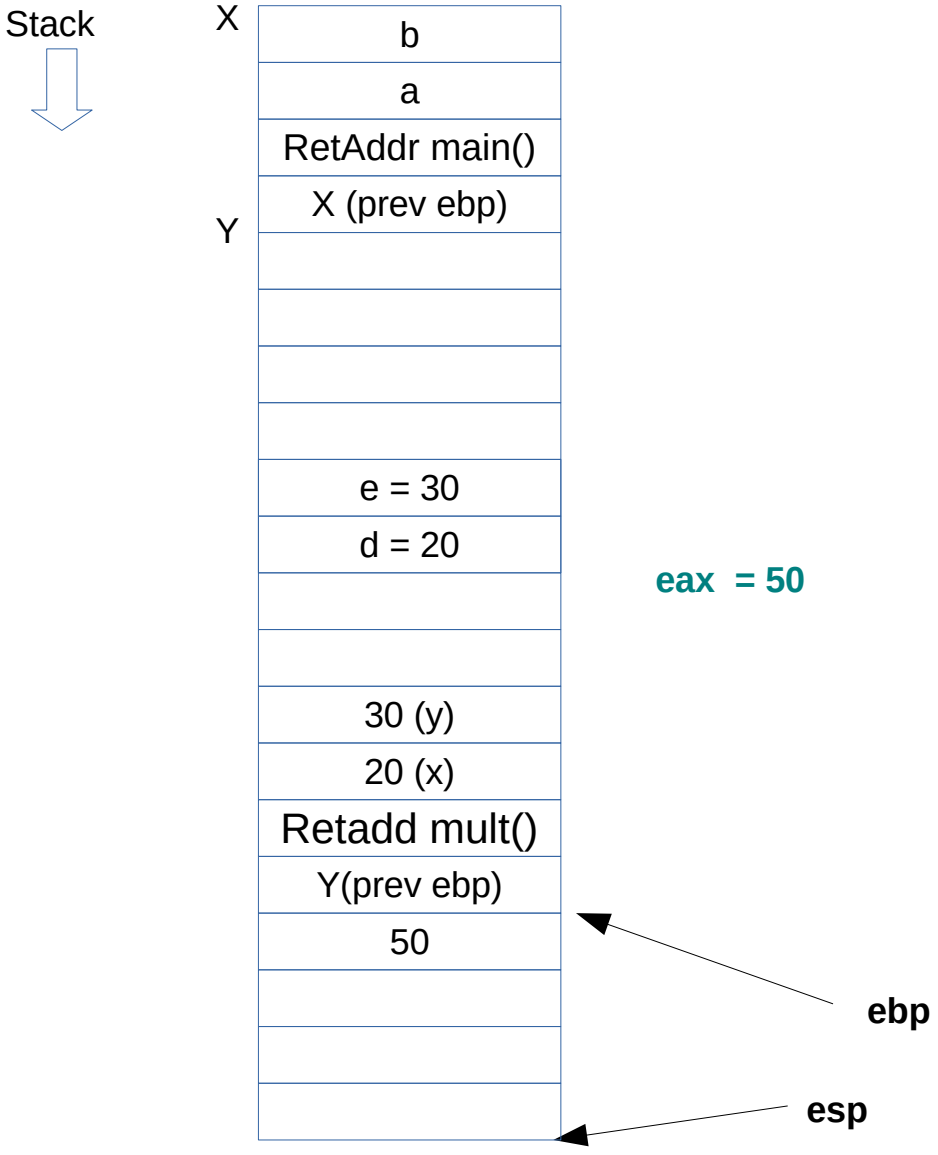
```
add:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    8(%ebp), %edx
        movl    12(%ebp), %eax
        addl    %edx, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        leave
        ret
```




```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e);
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

edx = 20
eax = 30
eax = eax + edx = 50

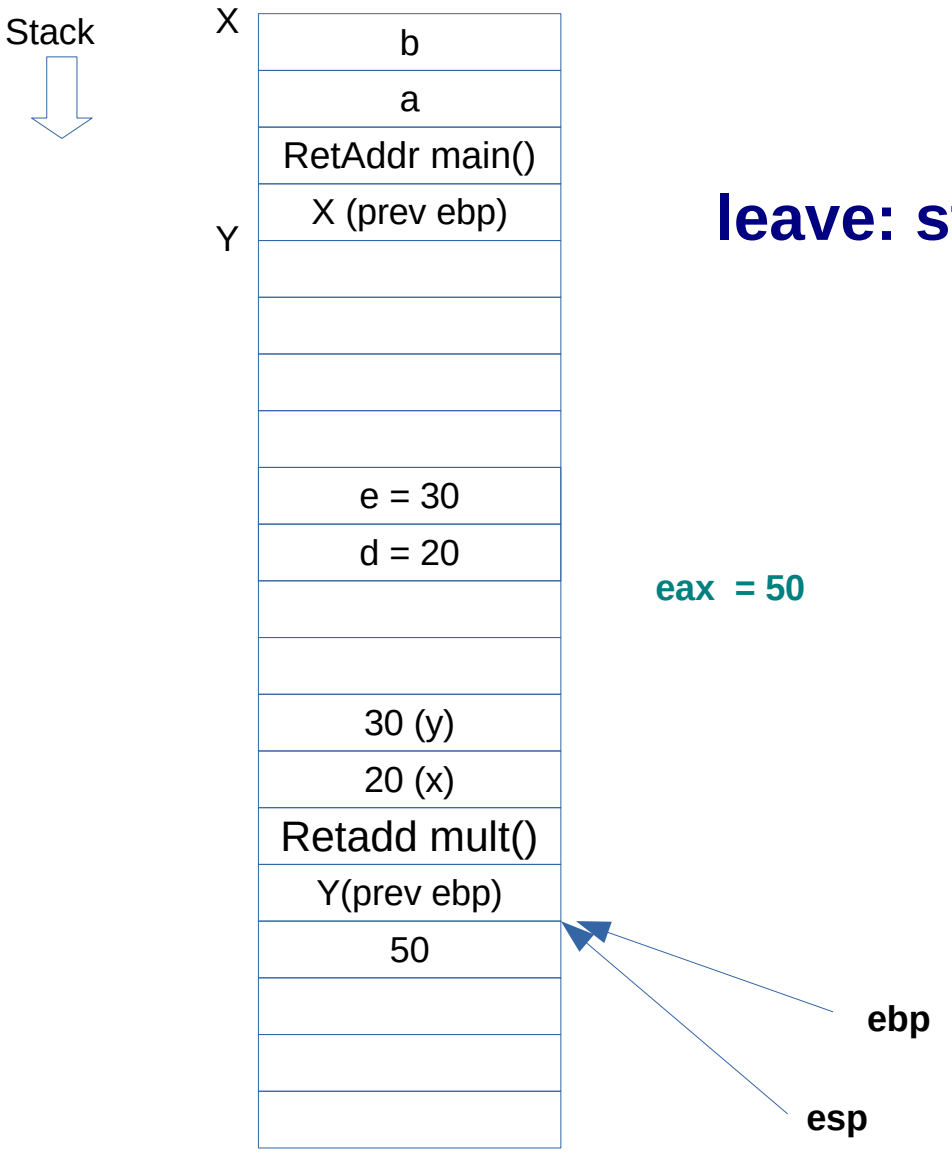
```
add:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    8(%ebp), %edx
        movl    12(%ebp), %eax
        addl    %edx, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        leave
        ret
```




```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e);
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

```
add:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    8(%ebp), %edx
        movl    12(%ebp), %eax
        addl    %edx, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        leave
        ret
```

Some redundant code generated here. Before “leave”. Result is in eax

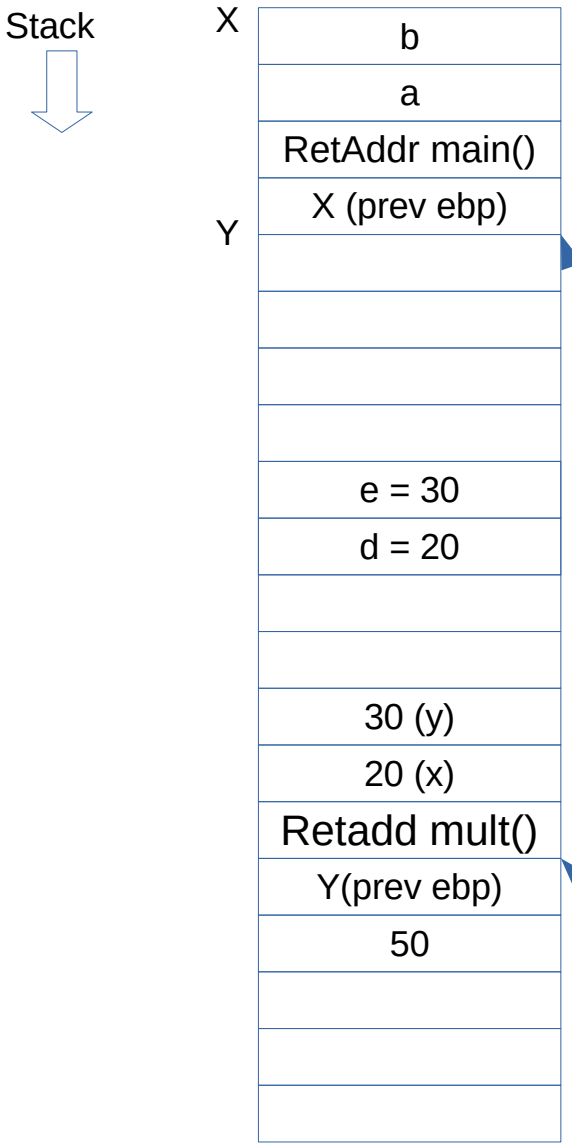


leave: step 1

eax = 50

```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e);
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

```
add:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    8(%ebp), %edx
        movl    12(%ebp), %eax
        addl    %edx, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        leave # # Set ESP to EBP, then pop EBP.
        ret
```

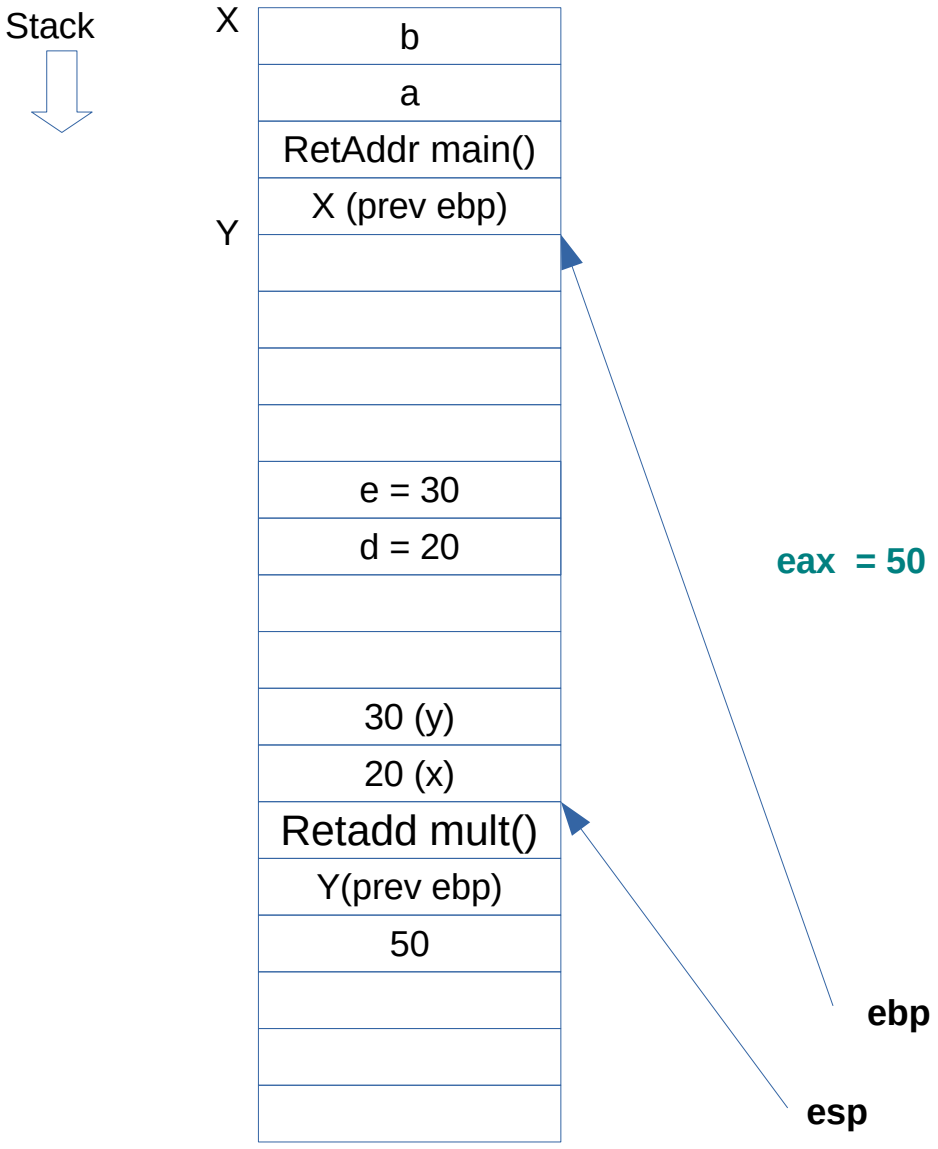


leave: step 2

eax = 50

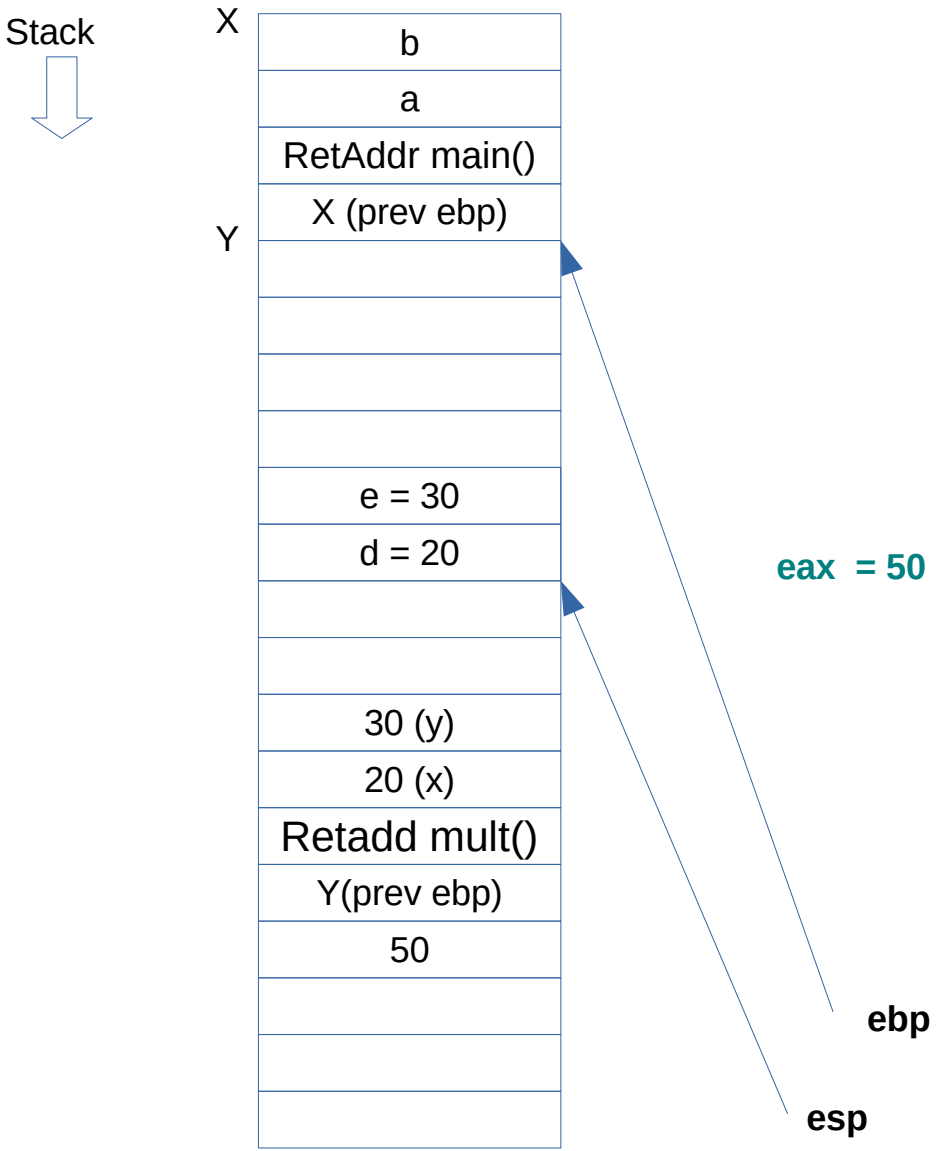
```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e);
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

```
add:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    8(%ebp), %edx
        movl    12(%ebp), %eax
        addl    %edx, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        leave # # Set ESP to EBP,
then pop EBP.
        ret
```



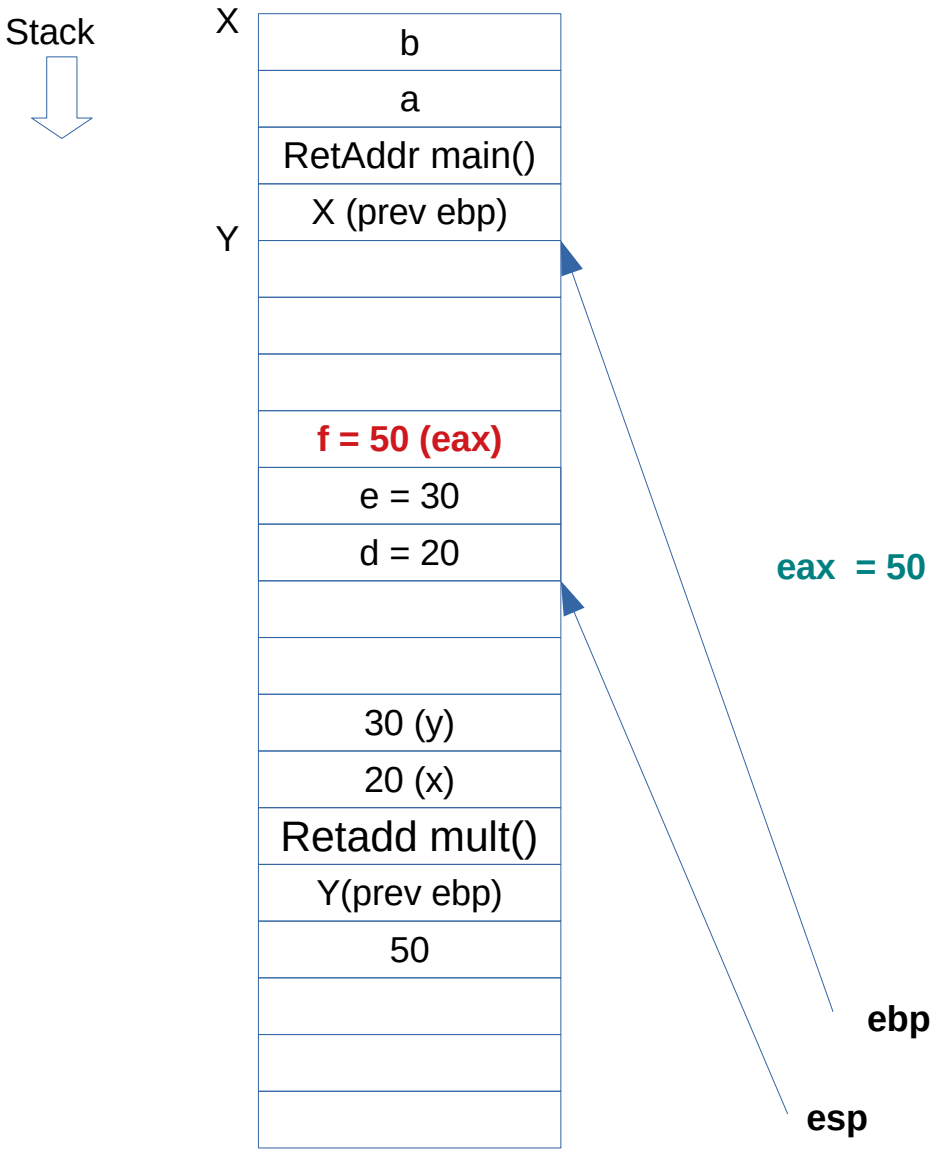

```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e); // here
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

```
add:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    8(%ebp), %edx
        movl    12(%ebp), %eax
        addl    %edx, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        leave # # Set ESP to EBP, then pop EBP.
        ret
```




```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e); // here
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

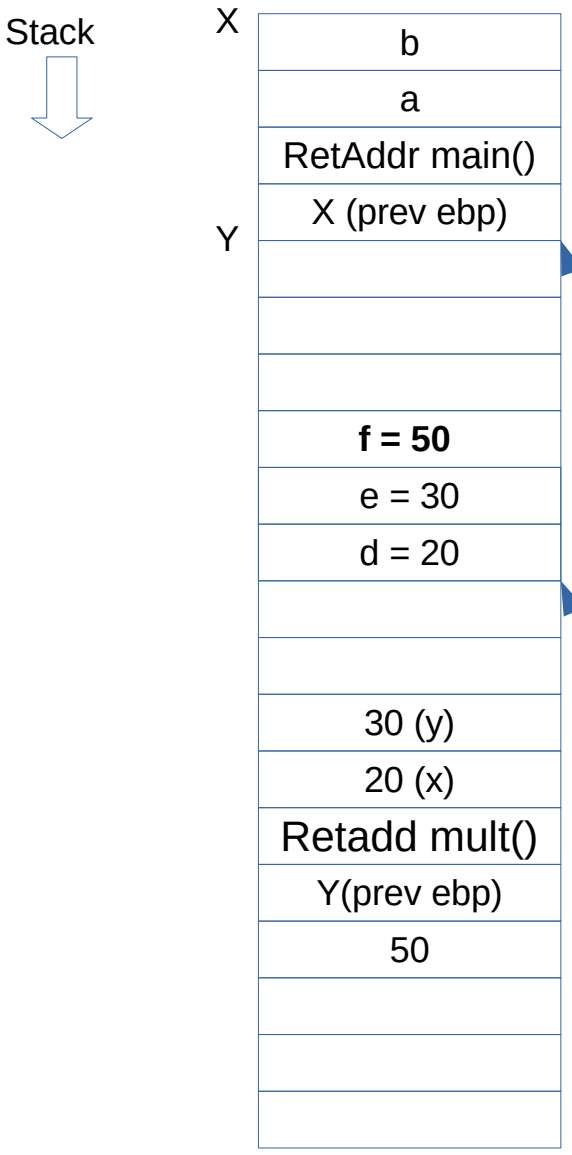
```
Mult:
        ....
          call    add
        addl    $16, %esp
        movl    %eax, -16(%ebp)
        movl    8(%ebp), %eax
        imull   12(%ebp), %eax
        movl    %eax, %edx
        movl    -16(%ebp), %eax
        addl    %edx, %eax
        movl    %eax, -12(%ebp)
        movl    -12(%ebp), %eax
        leave
        ret
```




```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e);
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

Mult:

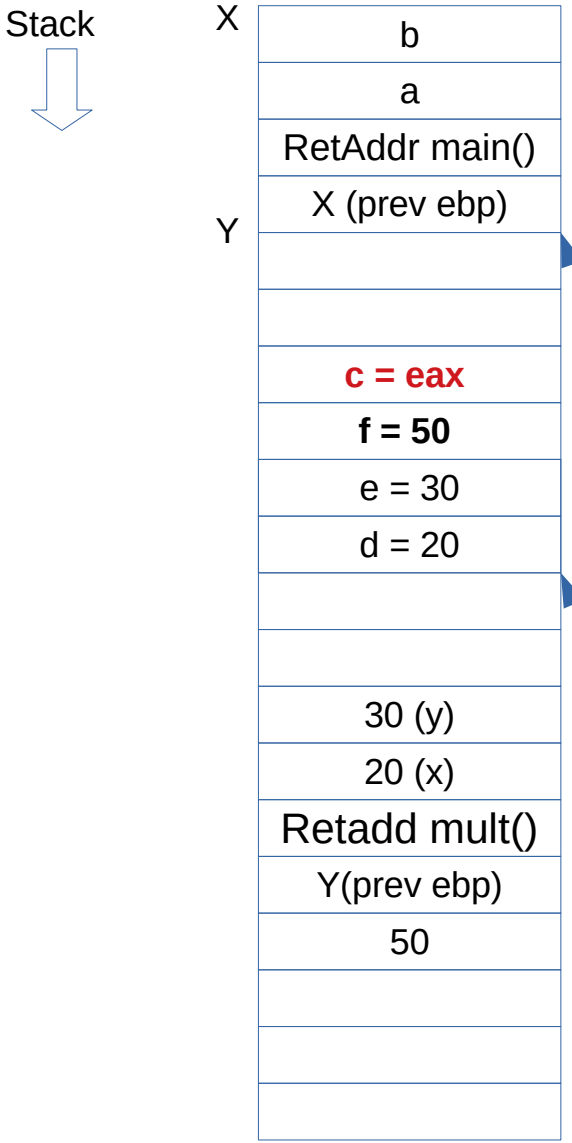
```
....
  call   add
addl   $16, %esp
movl    %eax, -16(%ebp)
movl   8(%ebp), %eax
imull  12(%ebp), %eax
movl    %eax, %edx
movl    -16(%ebp), %eax
addl   %edx, %eax
movl    %eax, -12(%ebp)
movl    -12(%ebp), %eax
leave
ret
```




```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e);
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

```
eax = a
eax = eax * b
edx = eax
eax = f
eax  = edx + eax
// eax = a*b + f
```

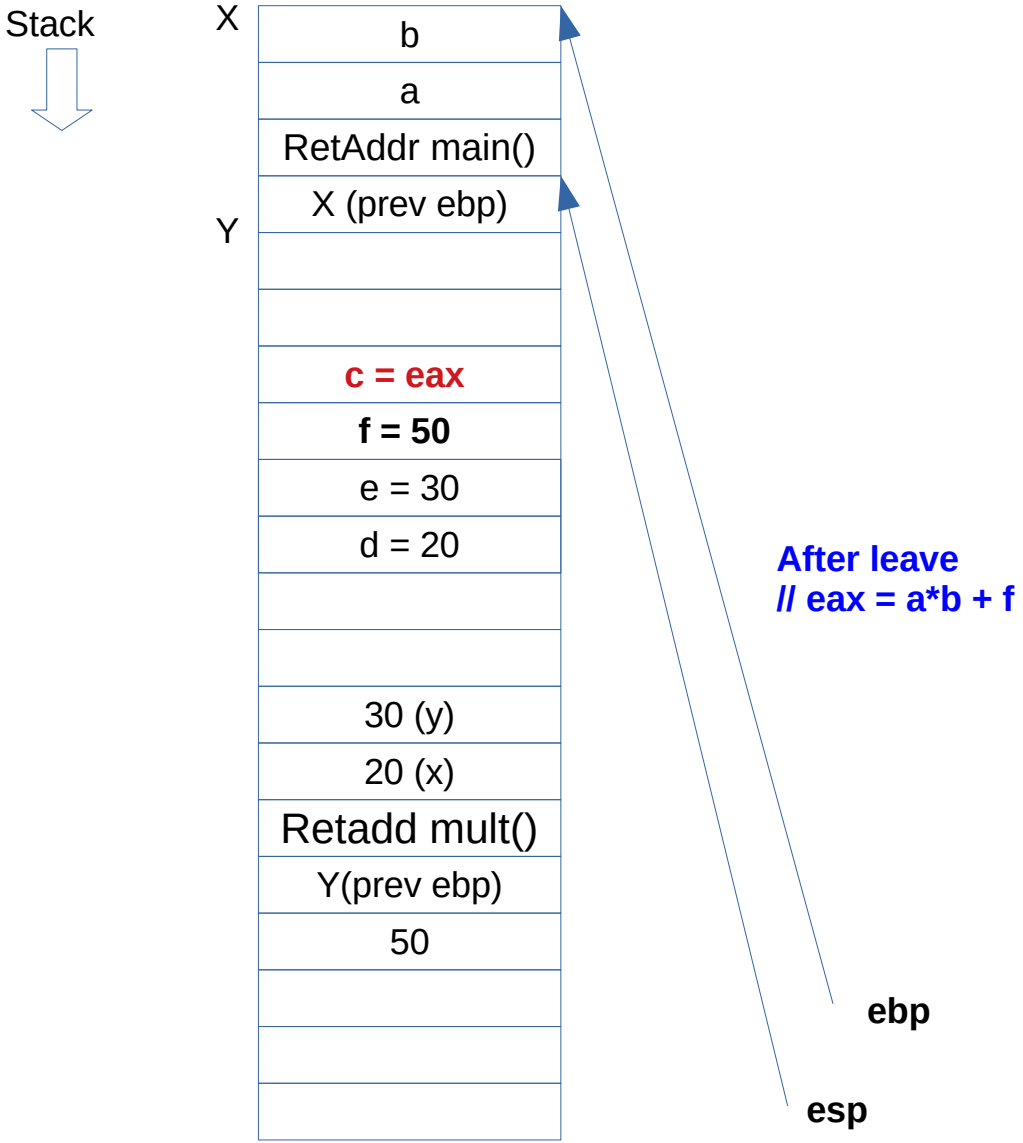
```
Mult:
        ....
          call    add
        addl    $16, %esp
        movl    %eax, -16(%ebp)
        movl    8(%ebp), %eax
        imull   12(%ebp), %eax
        movl    %eax, %edx
        movl    -16(%ebp), %eax
        addl    %edx, %eax
        movl    %eax, -12(%ebp)
        movl    -12(%ebp), %eax
        leave
        ret
```




```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e);
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

Mult:

```
....
  call   add
addl   $16, %esp
movl    %eax, -16(%ebp)
movl   8(%ebp), %eax
imull  12(%ebp), %eax
movl    %eax, %edx
movl    -16(%ebp), %eax
addl   %edx, %eax
movl    %eax, -12(%ebp)
movl   -12(%ebp), %eax
leave
ret
```




```
int mult(int a, int b) {
        int c, d = 20, e = 30, f;
        f = add(d, e);
        c = a * b + f;
        return c;
}
int add(int x, int y) {
        int z;
        z = x + y;
        return z;
}
```

```
Mult:
        ....
          call    add
        addl    $16, %esp
        movl    %eax, -16(%ebp)
        movl    8(%ebp), %eax
        imull   12(%ebp), %eax
        movl    %eax, %edx
        movl    -16(%ebp), %eax
        addl    %edx, %eax
        movl    %eax, -12(%ebp)
        movl    -12(%ebp), %eax
        leave
        ret
```

Lessons

- **Calling function (caller)**
  - **Pushes arguments on stack , copies values**
- **On call**
  - **Return IP is pushed**
- **Initially in called function (callee)**
  - **Old ebp is pushed**
  - **ebp = stack**
  - **Stack is decremented to make space for local variables**

Lessons

- **Before Return**
  - **Ensure that result is in ‘eax**
- **On Return**
  - **stack = ebp**
  - **Pop ebp (ebp = old ebp)**
- **On ‘ret’**
  - **Pop ‘return IP’ and go back in old function**

Lessons

- **This was a demonstration for a**
  - **User program, compiled with GCC, On Linux**
  - **Followed the conventions we discussed earlier**
- **Applicable to**
  - **C programs which work using LIFO function calls**
- **Compiler can’t generate code using this mechanism for**
  - **Functions like fork(), exec(), scheduler(), etc.**
  - **Boot code of OS**

Notes on reading xv6 code

Abhijit A. M.
abhijit.comp@coep.ac.in

Credits:
xv6 book by Cox, Kaashoek, Morris
Notes by Prof. Sorav Bansal

Introduction to xv6

Structure of xv6 code

Compiling and executing xv6 code

About xv6

- Unix Like OS
- Multi tasking, Single user
- On x86 processor
- Supports some system calls
- Small code, 7 to 10k
- Meant for learning OS concepts
- No : demand paging, no copy-on-write fork, no shared-memory, fixed size stack for user programs

Use cscope and ctags with VIM

- Go to folder of xv6 code and run

```
cscope -q *.[chS]
```

- Also run

```
ctags *.[chS]
```

- Now download the file http://cscope.sourceforge.net/cscope_maps.vim as `.cscope_maps.vim` in your ~ folder
- And add line "`source ~/.cscope_maps.vim`" in your `~/.vimrc` file
- Read this tutorial http://cscope.sourceforge.net/cscope_vim_tutorial.html

Use call graphs (using doxygen)

- Doxygen – a documentation generator.
- Can also be used to generate “call graphs” of functions
- Download xv6
- Install doxygen on your Ubuntu machine.
- cd to xv6 folder
- Run “doxygen -g doxyconfig”
  - This creates the file “doxyconfig”

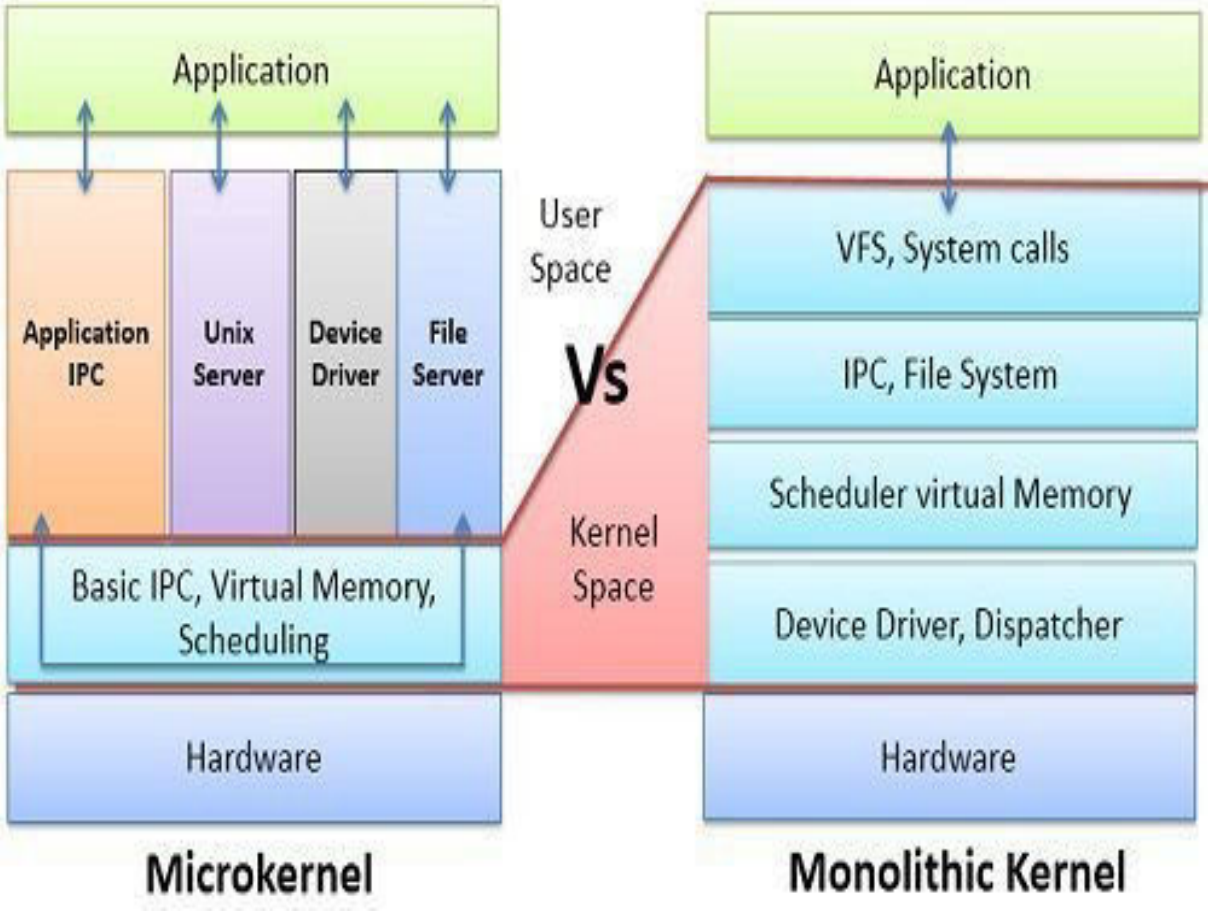
Use call graphs (using doxygen)

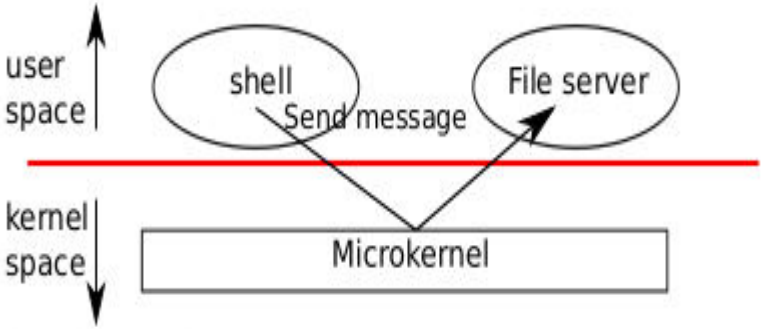
- Create a folder “doxygen”
- Open “doxyconfig” file and make these changes.

```
PROJECT_NAME           = "XV6"
OUTPUT_DIRECTORY       = ./doxygen
CREATE_SUBDIRS         = YES
EXTRACT_ALL            = YES
EXCLUDE                = usertests.c cat.c yes.c echo.c forktest.c
grep.c init.c kill.c ln.c ls.c mkdir.c rm.c sh.c stressfs.c wc.c
zombie.c
CALL_GRAPH             = YES
CALLER_GRAPH           = YES
```

- Now run “doxygen doxyconfig”
- Go to “doxygen”/html and open “firefox index.html” --> See call graphs in files -> any file

Xv6 follows monolithic kernel approach





qemu

- A virtual machine manager, like Virtualbox
- Qemu provides us
  - BIOS
  - Virtual CPU, RAM, Disk controller, Keyboard controller
  - IOAPIC, LAPIC
- Qemu runs xv6 using this command

```
qemu -serial mon:stdio -drive
file=fs.img,index=1,media=disk,format=raw -drive
file=xv6.img,index=0,media=disk,format=raw -smp 2 -
m 512
```

  - Invoked when you run “make qemu”

qemu

- **Understanding qemu command**
  - **-serial mon:stdio**
    - **the window of xv6 is also multiplexed in your normal terminal.**
    - **Run “make qemu”, then Press “Ctrl-a” and “c” in terminal and you get qemu prompt**
  - **-drive file=fs.img,index=1,media=disk,format=raw**
    - **Specify the hard disk in “fs.img”, accessible at first slot in IDE(or SATA, etc), as a “disk” , with “raw” format**
  - **-smp 2**
    - **Two cores in SMP mode to be simulated**
  - **-m 512**
    - **Use 512 MB ram**

About files in XV6 code

- `cat.c echo.c forktest.c grep.c init.c kill.c ln.c ls.c mkdir.c rm.c sh.c stressfs.c usertests.c wc.c yes.c zombie.c`
  - User programs for testing xv6
- `Makefile`
  - To compile the code
- `dot-bochsrc`
  - For running with emulator bochs

About files in XV6 code

- `bootasm.S  entryother.S  entry.S  initcode.S  swtch.S  trapasm.S  usys.S`
  - Kernel code written in Assembly. Total 373 lines
- `kernel.ld`
  - Instructions to Linker, for linking the kernel properly
- `README  Notes LICENSE`
  - Misc files

Using Makefile

- **make qemu**
  - **Compile code and run using “qemu” emulator**
- **make xv6.pdf**
  - **Generate a PDF of xv6 code**
- **make mkfs**
  - **Create the mkfs program**
- **make clean**
  - **Remove all intermediary and final build files**

Files generated by Makefile

- `.o files`
  - Compiled from each .c file
  - No need of separate instruction in Makefile to create .o files
  - _%: %.o $(ULIB) line is sufficient to build each .o for a _xyz file
-

Files generated by Makefile

- **asm files**
  - **Each of them has an equivalent object code file or C file. For example**

```
bootblock: bootasm.S bootmain.c
        $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c
bootmain.c
        $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c
bootasm.S
        $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o
bootblock.o bootasm.o bootmain.o
        $(OBJDUMP) -S bootblock.o > bootblock.asm
        $(OBJCOPY) -S -O binary -j .text bootblock.o
bootblock
        ./sign.pl bootblock
```

Files generated by Makefile

- `_ln, _ls, etc`
  - **Executable user programs**
  - **Compilation process is explained after few slides**

Files generated by Makefile

- **xv6.img**
  - **Image of xv6 created**

```
xv6.img: bootblock kernel
        dd if=/dev/zero of=xv6.img count=10000
        dd if=bootblock of=xv6.img conv=notrunc
        dd if=kernel of=xv6.img seek=1 conv=notrunc
```

Files generated by Makefile

- **bootblock**

```
bootblock: bootasm.S bootmain.c
        $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c
bootmain.c
        $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c
bootasm.S
        $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o
bootblock.o bootasm.o bootmain.o
        $(OBJDUMP) -S bootblock.o > bootblock.asm
        $(OBJCOPY) -S -O binary -j .text bootblock.o
bootblock
        ./sign.pl bootblock
```

Files generated by Makefile

kernel

```
kernel: $(OBJS) entry.o entryother initcode kernel.ld
        $(LD) $(LDFLAGS) -T kernel.ld -o kernel  entry.o $(OBJS) -b binary initcode entryother
        $(OBJDUMP) -S kernel > kernel.asm
        $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

Files generated by Makefile

- **fs.img**
  - A disk image containing user programs and README

```
fs.img: mkfs README $(UPROGS)
        ./mkfs fs.img README $(UPROGS)
```

- **.sym files**
  - Symbol tables of different programs
  - E.g. for file “kernel”

```
$(OBJDUMP) -t kernel | sed '1,/SYMBOL
TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

Size of xv6 C code

- **wc *[ch] | sort -n**
  - **10595 34249 278455 total**
  - **Out of which**
    - **738 4271 33514 dot-bochsrc**
- **wc cat.c echo.c forktest.c grep.c init.c kill.c ln.c ls.c mkdir.c rm.c sh.c stressfs.c usertests.c wc.c yes.c zombie.c**
  - **2849 6864 51993 total**
- So total code is 10595 – 2849 – **738 = 7008 lines**

List of commands to try (in given order)

usertests # Runs lot of tests and takes upto 10 minutes to run

stressfs # opens , reads and writes to files in parallel

ls # out put is filetyep, inode number, type

cat README

ls;ls

cat README | grep BUILD

echo hi there

echo hi there | grep hi

echo "hi there

List of commands to try (in this order)

echo README | grep Wa

echo README | grep Wa | grep ty # does not work

cat README | grep Wa | grep bl # works

ls > out # takes time!

mkdir test

cd test

ls # fails

ls ../ # works from inside test

cd # fails

cd / # works

wc README

rm out

ls . test # listing both directories

ln cat xyz; ls

rm xyz; ls

User Libraries: Used to link user land programs

- **Ulib.c**
  - **Strcpy, strcmp,strlen, memset, strchr, stat, atoi, memove**
  - **Stat uses open()**
- **Usys.S -> compiles into usys.o**
  - **Assembly code file. Basically converts all calls like open() (e.g. used in ulib.c) into assembly code using "int" instruction.**

  Run following command see the last 4 lines in the output

```
objdump -d usys.o

00000048 <open>:
  48:   b8 0f 00 00 00          mov    $0xf,%eax
  4d:   cd 40                   int    $0x40
  4f:   c3                      ret
```

User Libraries: Used to link user land programs

- printf.c
  - Code for printf()!
  - Interesting to read this code.
    - Uses variable number of arguments. Normal technique in C is to use va_args library, but here it uses pointer arithmetic.
  - Written using two more functions: printint() and putc() - both call write()
    - Where is code for write()?

User Libraries: Used to link user land programs

- **umalloc.c**
  - **This is an implementation of malloc() and free()**
  - **Almost same as the one done in “The C Programming Language” by Kernighan and Ritchie**
  - **Uses sbrk() to get more memory from xv6 kernel**

Understanding the build process in more details

- **Run**

  make qemu | tee make-output.txt

- **You will get all compilation commands in make-output.txt**

Compiling user land programs

Normally when you compile a program on Linux

You compile it for the same ‘target’ machine (= CPU + OS)

The compiler itself runs on the same OS

To compile a user land program for xv6, we don’t have a compiler on xv6,

So we compile the programs (using make, cc) on Linux , for xv6

Obviously they can’t link with the standard libraries on Linux

Compiling user land programs

```
ULIB = ulib.o usys.o printf.o umalloc.o

_%: %.o $(ULIB)
        $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^
        $(OBJDUMP) -S $@ > $*.asm
        $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d;
s/ .* / /; /^$$/d' > $*.sym
```

$@ is the name of the file being generated

$^ is dependencies . i.e. $(ULIB) and %.o in this case

Compiling user land programs

```
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing
-O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-
pointer -fno-stack-protector -fno-pie -no-pie   -c -o
cat.o cat.c

ld -m    elf_i386 -N -e main -Ttext 0 -o _cat cat.o
ulib.o usys.o printf.o umalloc.o

objdump -S _cat > cat.asm

objdump -t _cat | sed '1,/SYMBOL TABLE/d;
s/ .* / /; /^$/d' > cat.sym
```

Compiling user land programs

Mkfs is compiled like a Linux program !

gcc -Werror -Wall -o mkfs mkfs.c

How to read kernel code ?

- **Understand the data structures**
  - **Know each global variable, typedefs, lists, arrays, etc.**
  - **Know the purpose of each of them**
- **While reading a code path, e.g. exec()**
  - **Try to ‘locate’ the key line of code that does major work**
  - **Initially (but not forever) ignore the ‘error checking’ code**
- **Keep summarising what you have read**
  - **Remembering is important !**
- **To understand kernel code, you should be good with concepts in OS , C, assembly, hardware**

Pre-requisits for reading the code

- **Understanding of core concepts of operating systems**
  - **Memory Management, processes, fork-exec, file systems, synchronization, x86 architecture, calling convention , computer organization**
- **2 approaches:**

  1) Read OS basics first, and then start reading xv6 code
  - **Good approach, but takes more time !**

  2) Read some basics, read xv6, repeat
  - **Gives a headstart, but you will always have gaps in your understanding of the code, until you are done with everything**
  - **We normally follow this approach**
- **Good knowledge of C, pointers, function pointers particularly**
  - **Data structures: doubly linked lists, queues, structures and pointers**