How a CPU executes programs

Ramin Sadre

Memory organization

- On computers and operating systems with virtual memory, each running process gets its own virtual address space
- When the program is loaded, the OS reserves blocks in main memory for
  - the code (also called “text”) of the program
  - the (static) data of the program (global constants and variables etc.)
  - same for dynamically loaded libraries (`.dll` on Windows, `.so` on Linux)
- Once the program has been started, it can allocate more blocks for dynamic data structures etc. with `new` or `alloc`

Memory organization: Example

- Let's assume a C program with 2 global 32-bit variables *i* and *j* and the instruction `i=i+j`

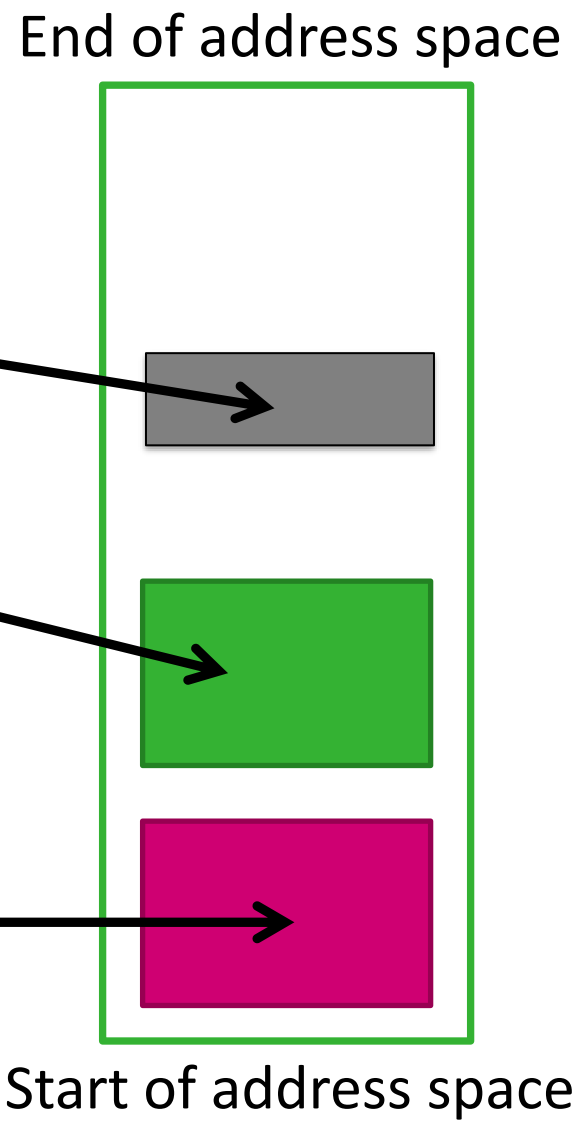
Some dynamically allocated memory

Statically allocated memory for data:

```
0x02000000 (4 bytes for i)
0x02000004 (4 bytes for j)
0x01000008 ...
```

Code:

```
0x01000000 load32  0x02000000,r1
0x01000006 load32  0x02000004,r2
0x0100000C add     r1,r2,r1
0x0100000E store32 r1,0x02000000
```



Memory vs registers

- A CPU has several **registers** = temporary data stores that are used to perform calculations etc.
- For the CPU, **variables** are just locations in main memory
- There is a special register that contains the address of the next instruction to be executed, called the instruction pointer (IP) or program counter (PC)
- After each instruction, the IP is moved further

```
IP = 0x0100006

0x01000000 load32  0x02000000,r1
0x01000006 load32  0x02000004,r2
0x0100000C add     r1,r2,r1
0x0100000E store32 r1,0x02000000
```

Variables in memory

- For the CPU, variables don’t have a structure. Memory is just a collection of 8/16/32/64-bit words

```
char str[16];
int i,j;
```

```
0x03000000   str[0]
0x03000001   str[1]
...
0x0300000F   str[15]
0x03000010   i
0x03000014   j
```

- For performance reasons, compilers sometimes align variables to 32-bit or 64-bit boundaries (or even re-order them)

```
char str[3];
int i;
```

```
0x03000000   str[0]
0x03000001   str[1]
0x03000002   str[2]
0x03000003   (unused)
0x03000004   i
```

C strings

- C is a language very close to the machine
- In C, strings are not objects but `char` arrays that are terminated with a 0-byte

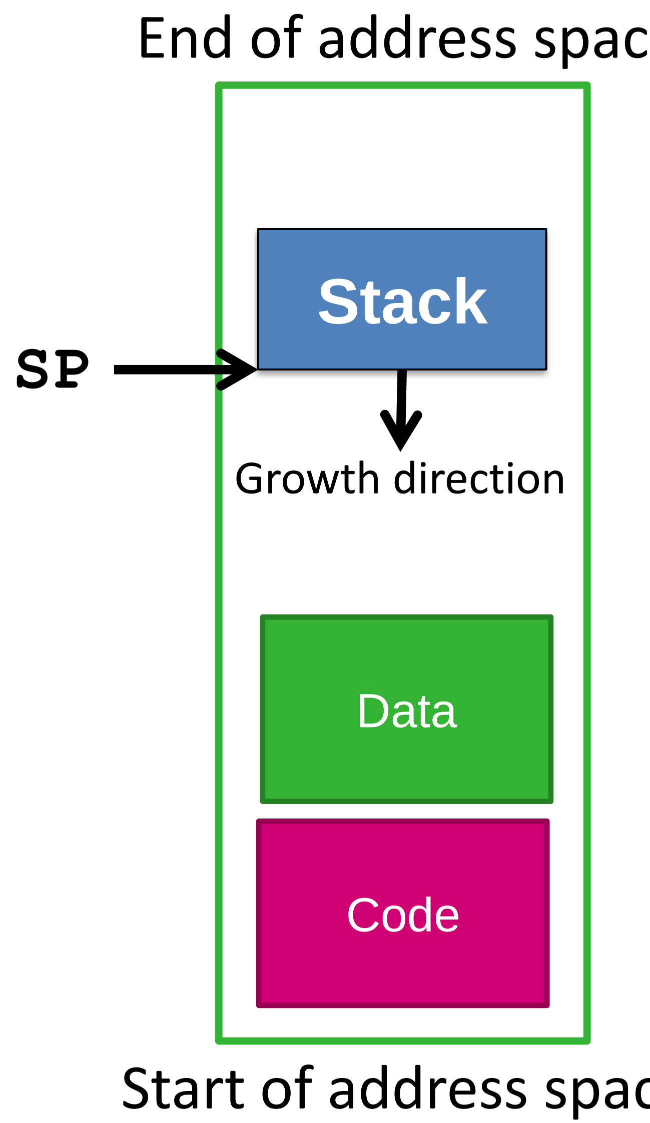
```
char str[8]="hello";
```

```
0x03000000  'h'
0x03000001  'e'
0x03000002  'l'
0x03000003  'l'
0x03000004  'o'
0x03000005  0
0x03000006  0
0x03000007  0
```

- (According to the C standard, the unused elements `str[6]` and `str[7]` are initialized with 0 by the compiler or at program start)

The stack

- CPUs maintain a special datastructure that simplifies the implementation of function calls: **the stack**
- The stack stores information about the called functions and holds their local variables and parameters
- Because it is not known in advance how much stack space a program needs it is put close to the end of the address space and grows therefore <u>downwards</u>
- A special register **SP** contains the address of the top of the stack



Function calls

- Lets imagine a function f with two parameters a and b and two local variables i and j
- We are currently in function g and want to call f

```
void f(int a, int b) {
    int i, j;
    ...
    return;
}

void g() {
    f(3,4);
    ...
}
```

call:
IP = 0x20000000

```
0x20000000 function f
...
...
0x20000030 return

0x30000000 push 4
0x30000004 push 3
0x30000008 call 0x20000000
0x3000000e ...
```

return:
IP = 0x3000000e

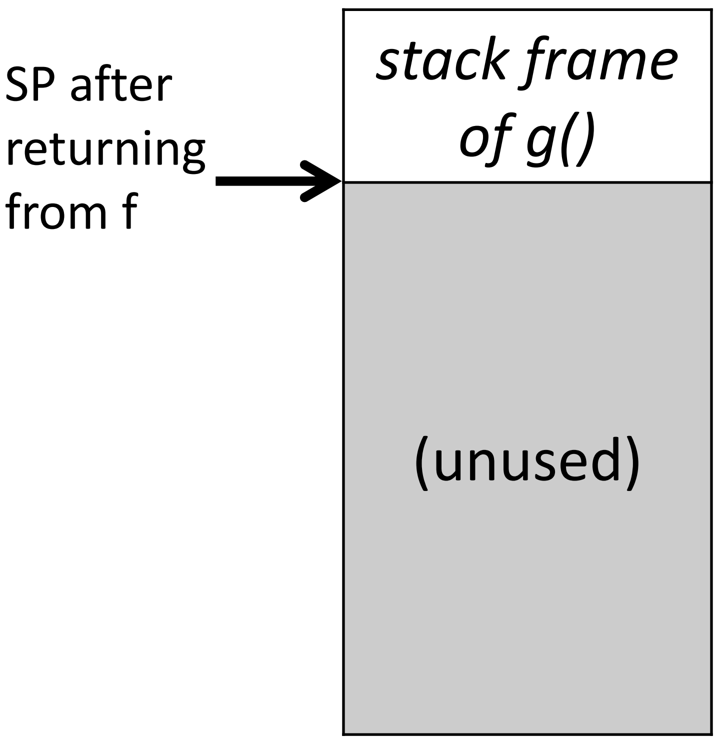
Stack frame

- When the function *f* is called, the following information (a *stack frame*) is put on the stack during runtime:

	stack frame of caller g()	*(higher addresses)*
SP before calling *f* →	4	Second parameter value
	3	First parameter value
	0x3000000e	The return address
	frame pointer	
	j	Space for local variable j
SP when inside function *f* →	i	Space for local variable i
	(unused)	*(lower addresses)*

Returning from a function

- When $f(3,4)$ returns to $g()$, the top stack frame is removed from the stack and the program execution continues at the instruction stored at the return address

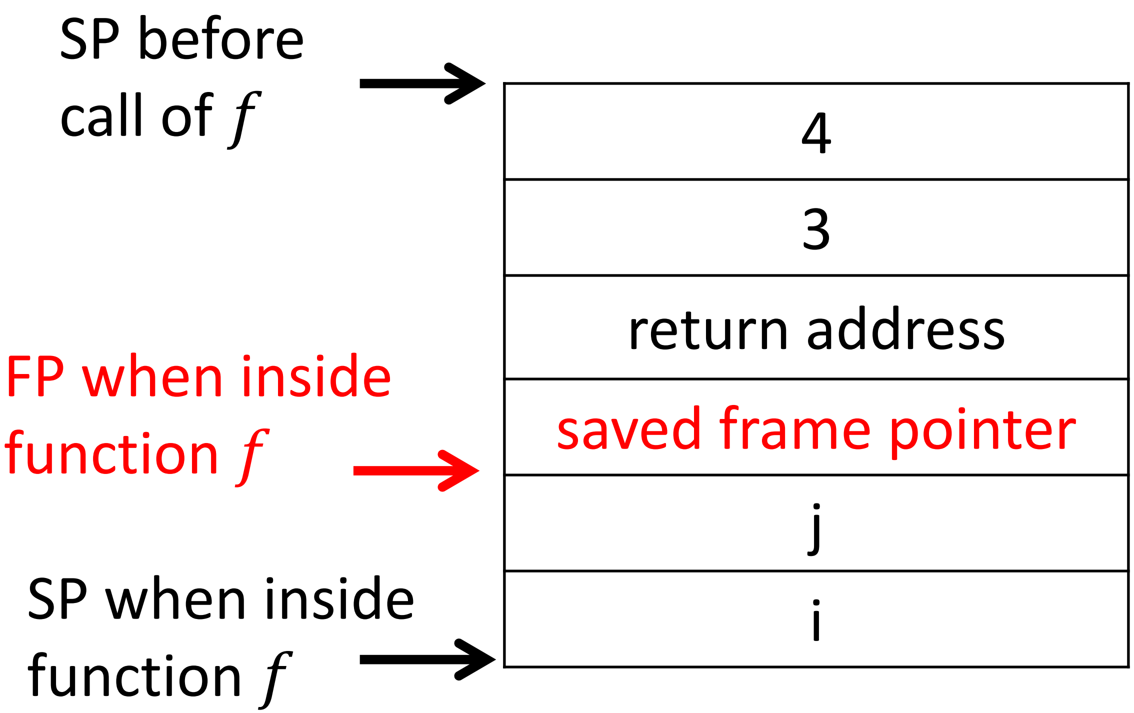



```
0x20000000 code for function f
...
...
0x20000030 return


0x30000000 push 4
0x30000004 push 3
0x30000008 call 0x20000000
0x3000000e ...
```

The frame pointer

- For convenience, many CPUs have a framepointer register FP that points at the end of the block with local variables
- Because the FP has to be restored when returning from a function, its previous value is also stored on the stack



The frame pointer (2)

- What is the frame pointer used for?
- It allows a function to easily calculate the addresses of its local variables and parameters
- Example with 32-bit variables and addresses:

4	Address: FP+12
3	Address: FP+8
return address	Address: FP+4
saved frame pointer	Address: FP+0
j	Address: FP-4
i	Address: FP-8

FP when inside function f →

SP when inside function f →

Example: Intel x86 32-bit CPU

- On Intel CPUs the stack pointer %esp and the framepointer %ebp (base pointer) are manually managed
- Calling the function f(3,4) from g():

```
pushl  4            ; push 4 onto the stack (4 bytes)
pushl  3            ; push 3 onto the stack (4 bytes)
call   0x20000000   ; put the return address on
                    ; the stack and jump to f()
addl   8,%esp       ; remove the parameter values
                    ; from the stack (8 bytes)
```

Example: Intel x86 32-bit CPU (2)

- Function f (starting at address 0x2000000):

```
pushl %ebp           ; save the framepointer on the stack
movl  %esp,%ebp      ; FP = SP
subl  8,%esp         ; make space for the local
                     ; variables i and j (8 bytes)
...
movl  %ebp,%esp      ; SP = FP. This effectively removes
                     ; the local variables from the stack
popl  %ebp           ; restores the old value of FP
ret                  ; jump back to the return address
                     ; and remove it from the stack.
```