Memory Model Tutorial III : function call frames

In this tutorial we explain how the memory is managed and organized when a function call happens. Our focus lays on the stack segment of the main memory.

4.a Frame of a Function Call

Suppose that function g is called by function f. To handle this call g the following items will be pushed on top of the control stack pointed by the register **SP** :

1. a place holder **$g** for the return value of the called function g,

and then the so called **function frame** (also called **stack frame** or simply **frame)** of g, containing all other needed data, cf. Fig. 4.1 :

2. the values of the arguments **arg$_i$** of the called function g, in inverse order,
3. a place holder **pc_caller** pointing to the next instruction to be executed when returning to the caller function f,
4. the values of the local variables **l_v$_i$** of the called function g,
5. a place holder **fp_caller** pointing to the function frame of the caller function f,
   and finally update the register **FP** which has to point to the just now created place holder fp_caller.

Note that (1) – (3) are executed in the caller function f environment, and (4) – (5) in the called function g environment. This environment change from f to g is done with the help of the symbol table.

Once this function frame is in place, the evaluation of the body of the called function g can start on top of the stack pointed by the register **SP**. The register **PC** points to the next instruction to be executed.

On the return to the caller function f, **$g** is updated to the return value of the called function g, the function frame is undone in the reverse order, and the registers **fp** and **pc** are updated to their “old” value. At this point **$g** is on the top of the stack.

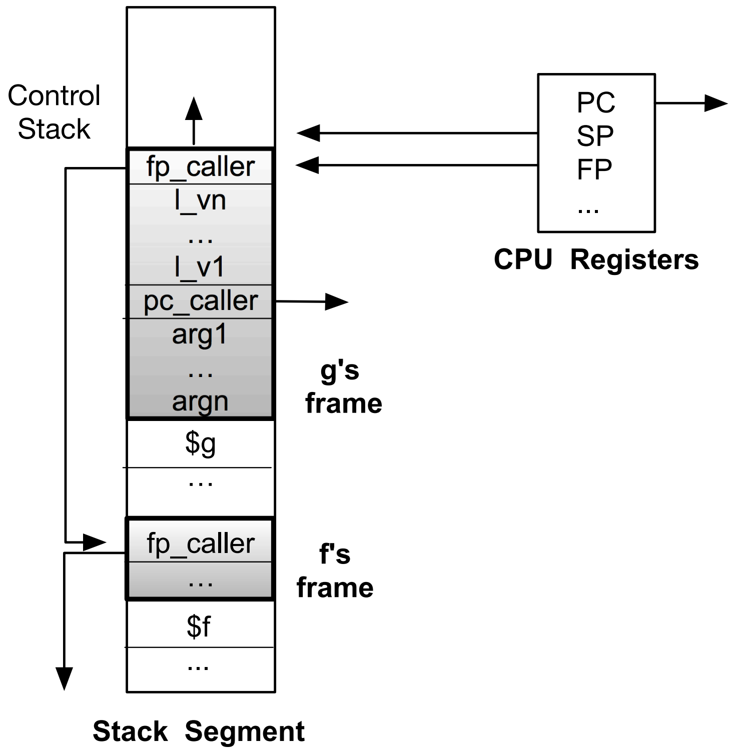

Fig. 4.1 Function call frames, inclusive CPU registers

4.b Example

Figure 4.3 illustrates the stack, data, heap and text segments of the program of Figure 4.2, assuming that the next instruction to be executed is the `'return'` of function f.

```
const float pi = 3.14;
int a1[];
int a2[2] = {1};

int f(int a; int b) {
   int x=3, y=4;
   return a+b+x+y;
}

int main() {
   int i=1, j=2, k;
   k=f(i,j);
   ...
}
```

Fig. 4.2 A simple C program with a function f and some global variables

Note that the evaluation of the expression `'k = f(i,j)'` will push on the control stack `'&k:'`, `'$f'`, and then f's frame. On f's `return`, the value on top of the stack will be copied to `'$f'`, and after f's frame has been popped, the next instruction will be executed, in this case `'='`, i.e. the value of `'$f'` will be copied to the memory pointed by `'&k:'`. The postfix `':'` indicates a constant value.

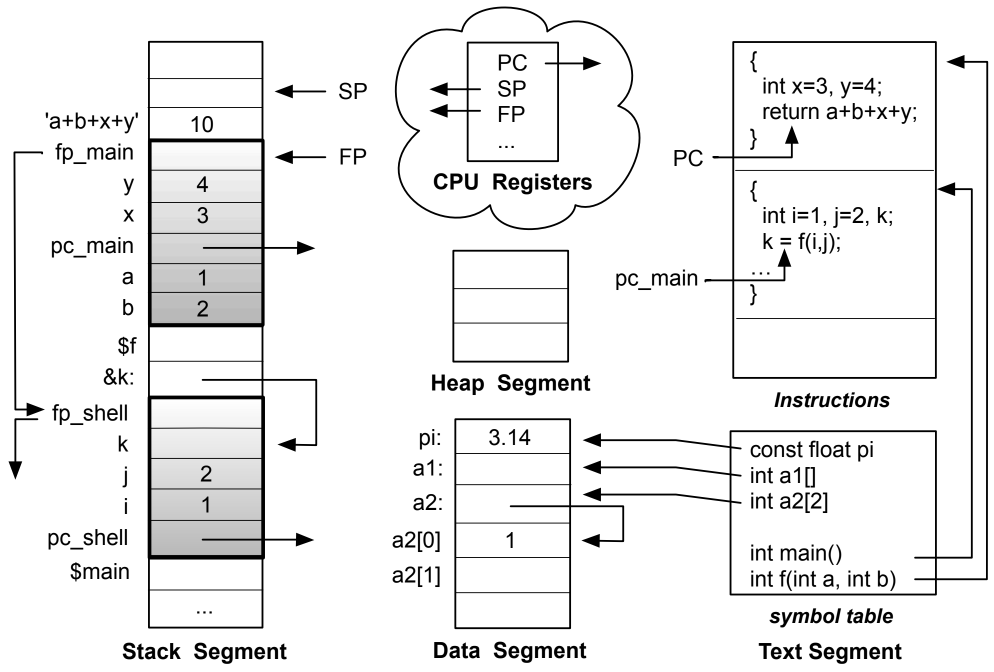

Fig. 4.3 Stack and text segments of a function call, inclusive CPU registers + some global variables in the data segment

We will often draw the stack segments by omitting the pc_x and fp_x pointers. We will call this a *simplified stack segment*.

In C, the heap segment is typically used by malloc().