Contents

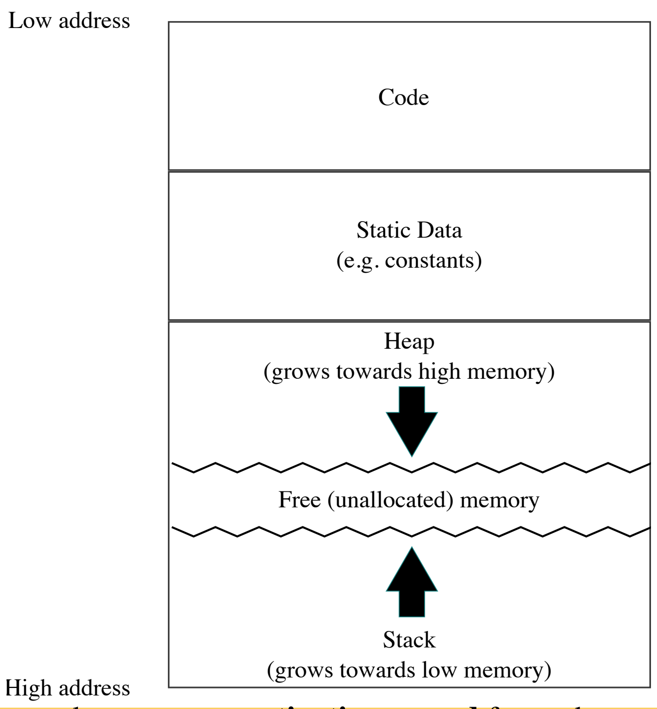
Introduction

In this set of notes we will consider:

- How storage is laid out at runtime.
- What information is stored for each method.
- What happens during method call and return for two different approaches to storage layout: static and stack allocation.

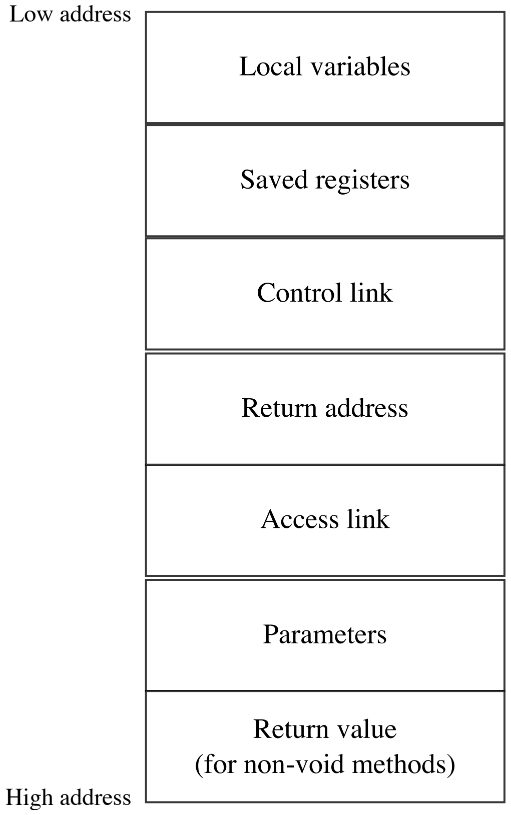
Storage Layout

There are many possible ways to organize memory. We will concentrate mainly on the standard Unix approach, illustrated below:



Low address
Code
Static Data
(e.g. constants)
Heap
(grows towards high memory)
Free (unallocated) memory
Stack
(grows towards low memory)
High address

Usually, the **stack** is used to store one **activation record** for each currently active method, and the **heap** is used for dynamically allocated memory (i.e., memory allocated as a result of using the *new* operator). An activation record is a data structure used to hold information relevant to one method call. The exact structure of an activation record depends both on the language in use and on the particular implementation; a typical organization is shown in the following picture (the individual fields will be discussed in some detail below and in the next set of notes).



As mentioned above, activation records are usually stored on the stack. A new record is pushed onto the stack when a method is called, and is popped when the method returns. However, for some languages, activation records may be stored in the heap (this might be done, for example, in a concurrent language, in which method calls do not obey the last-in-first-out protocol of a stack) or in the static data area. We will briefly consider the latter approach, then look at the most common case of stack allocation. In both cases, we will consider what must be done when a method is called, when it starts executing, and when it returns.

Static Allocation

Some old implementations of Fortran used this approach: there is no heap or stack, and all allocation records are in the static data area, one per method. This means that every time a method is called, its parameters and local variables are stored in the same locations (which are known at compile time). This approach has some advantages and disadvantages when compared with stack or heap allocation of activation records:

ADVANTAGES

- + fast access to all names (e.g., no need to compute the address of a variable at runtime)
- + no overhead of stack/heap manipulation

DISADVANTAGES

- - no recursion
- - no dynamic allocation

Using this approach, when a method is called, the ***calling method***:

- Copies each argument into the corresponding parameter's space in the called method's activation record (AR).
- May save some registers (in its own AR).
- Performs a "Jump & Link": Jump to the first instruction of the called method, and put the address of the next instruction after the call (the return address) into the special RA

register (the "return address" register).

The ***called*** method:

- Copies the return address from RA into its AR's return-address field.
- May save some registers (in its AR).
- May initialize local data.

When the called method is ready to return, it:

- Restores the values of any registers that it saved.
- Jumps to the address that it saved in its AR's return-address field.

Back in the calling method, the code that follows that call does the following:

- Restores any registers that it saved.
- If the called method was non-void (returned a value), put the return value (which may be in a special register or in the AR of the called method) in the appropriate place. For example, if the code was `x = f();`, then the return value should be copied into variable x.

TEST YOURSELF #1

Assume that static allocation is used, and that each activation record contains local variables, parameters, the return address, and (for non-void methods) the return value. Trace the execution of the following code by filling in the appropriate fields of the activation records of the three methods. Also think about where the string literals would be stored.

```
void error(String name, String msg) {
  System.out.println("ERROR in method " + name + ": " + msg);
}

int summation(int max) {
  int sum = 1;
  for (int k=1; k<=max; k++) {
    sum += k;
  }
  return sum;
}

void main() {
  int x = summation(3);
  if (x != 6) error("main", "bad value returned by summation");
}
```

solution

Stack Allocation

Stack allocation is used to implement most modern programming languages. The basic idea is that:

- Each time a method is called, a new AR (also called a **stack frame**) is pushed onto the stack.
- The AR is popped when the method returns.
- A register (SP for "stack pointer") points to the top of the stack.

- Another register (FP for "frame pointer") points to a fixed item (such as the return address or the access link) in the current method's AR.

When a method is called, the calling method:

- May save some registers (in its own AR).
- Pushes the parameters onto the stack (into space that is shared with the called method's AR).
- If the language allows nested methods, may set up the access link; this means pushing the appropriate value -- more on that in the next set of notes -- onto the stack.
- Does a "Jump & Link" -- jumps to the 1st instruction of the called method, and puts the address of the next instruction (the one after the call) into register RA.

The called method:

- Pushes the return address (from RA) onto the stack (into its AR's "return address" field).
- Pushes the old FP into its AR's "control link" field.
- Sets the FP to point to the appropriate place in its AR (to the "access link" field if there is one; otherwise, to the "return-address" field). The address of that field is computed as follows: SP + (size of "control link" field) + (size of "return address" field) + (size of "access link" field). All of these sizes are computed at *compile* time. (Note that values are *added* to the SP because we are assuming that "lower" on the stack means a *higher* address.)
- May save some registers (by pushing them onto the stack).
- Sets up the "local data" fields. This may involve pushing actual values if the locals are initialized as part of their declarations, or it may just involve subtracting their total size from the SP.

When the method returns, it:

- Restores the values of any saved registers.
- Loads the return address into register RA (from the AR).
- Restores the old stack pointer (SP = FP).
- Restores the old frame pointer (FP = saved FP, i.e., the value in the control-link field).
- Return (jump to the address in register RA).

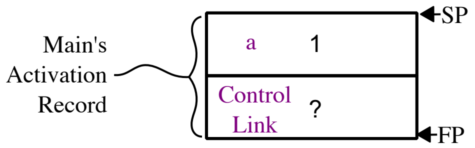
Example: Activation Records

Consider the following code:

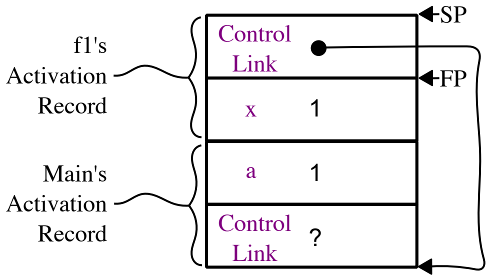
```
void f2(int y) {
  f1(y);
}

void f1(int x) {
  if (x > 0) f2(x-1);
}

main() {
  int a = 1;
  f(1);
}
```

The following pictures show the activation records on the stack at different points during the code's execution (only the control link, parameter, and local variable fields are shown).

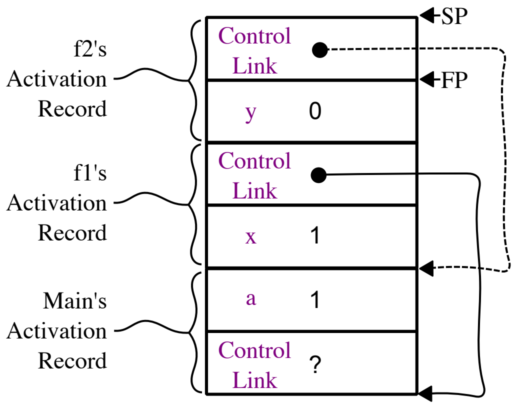
1. When the program starts:



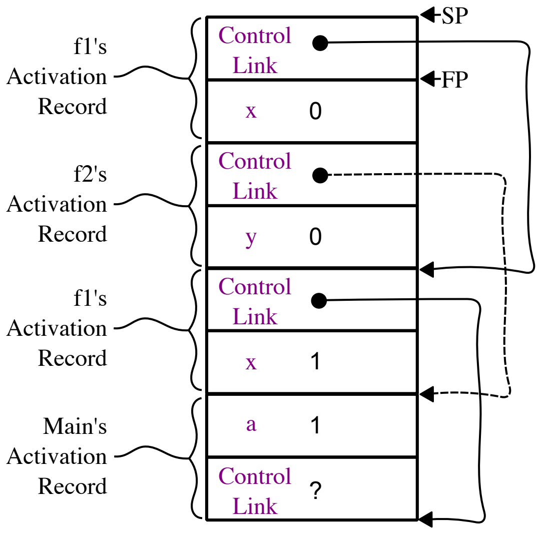
2. After main calls f1:



3. After f1 calls f2:



4. After f2 calls f1:



After this, f1 returns (and its AR is popped), then f2 returns, then the first call to f1 returns, then the whole program ends.

TEST YOURSELF #2

Assume that stack allocation is used. Trace the execution of the following code by filling in the local variables, parameters, and control link fields of the activation records (recall that dynamically allocated storage is stored in the heap, not on the stack).

```
void init(int[] A, int len) {
  for (int k=1; k<len; k++) {
    A[k] = k;
  }
}

void main() {
  int[] x = new int[3];
  init(x, 3);
}
```

solution
