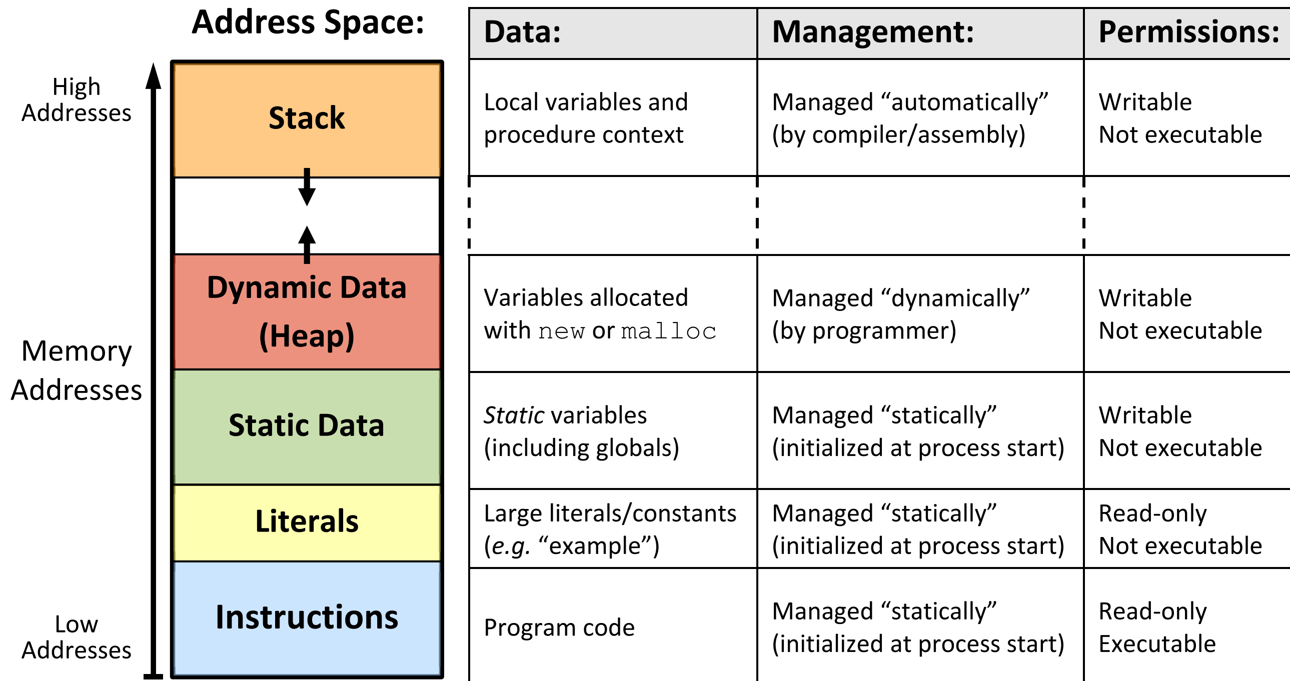
CSE 351 Lecture 11 – The Stack & Procedures

Memory Layout

Our memory address space is laid out in a particular way to help optimize our use of data:



Data:	Management:	Permissions:
Local variables and procedure context	Managed "automatically" (by compiler/assembly)	Writable Not executable
Variables allocated with `new` or `malloc`	Managed "dynamically" (by programmer)	Writable Not executable
Static variables (including globals)	Managed "statically" (initialized at process start)	Writable Not executable
Large literals/constants (*e.g.* "example")	Managed "statically" (initialized at process start)	Read-only Not executable
Program code	Managed "statically" (initialized at process start)	Read-only Executable

The ***stack*** is the section of memory that takes up the highest useable addresses and holds local variables and other procedure context (more details in the "Stack Frames" section below). As the arrow indicates, it "grows" downwards: the stack occupies lower addresses as it allocates more space.

The Stack Pointer and Manipulation

The end/"top" of the stack is indicated by the current address stored in the ***stack pointer*** (`%rsp`) – as `%rsp` is manipulated, the conceptual end of the stack will move with it (*i.e.* the addresses < `Reg[rsp]` are not considered part of the stack). This makes `%rsp` a useful reference point. For example, to access a local variable stored 8 bytes from the end of the stack, we can use the memory operand `8(%rsp)`.

The stack pointer can be changed directly via `subq` (for allocation) and `addq` (for deallocation) instructions, but note that this does not affect any of the *data* in memory, just which data are considered part of the stack.

We can also transfer data to and from the stack using the **`push`** *src* and **`pop`** *dst* instructions. `push` will (1) decrement `%rsp` and then (2) copy the data from the source operand into the newly-allocated space. `pop` will (1) copy the data from `%rsp` into the destination operand and then (2) increment `%rsp` to deallocate that space.

x86-64 Procedure Calling Conventions

The ***calling conventions*** are the established set of rules to guarantee that procedures can properly pass data and control to one another. We will learn the conventions for x86-64; they will be similar, but not exactly the same, in other ISAs. When describing one procedure calling another, we will call the procedure doing the calling the ***caller*** and the procedure being called the ***callee***. Because a procedure can be called from multiple places in code, we need a way to indicate how to return to the caller. We do this by storing a ***return address*** on the stack, which is address of the caller's next instruction to execute.

To pass control to another procedure we use the **`call`** *label* instruction. This will automatically push the return address, which is the address of the instruction *after* `call`, onto the stack before updating the program counter to the address of the specified label. When a procedure wants to return control, it uses the **`ret`** instruction to pop the return address off of the stack and into `%rip`. Note that this means that `%rsp` *must* be pointing at the return address before calling `ret` to avoid unexpected/bad behavior!

To pass ***arguments***, the first six must be placed in the following registers in order: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9`. A mnemonic that may help you remember this is "Diane's Silk Dress Cost $8 9." All additional arguments must be placed on the stack in reverse order (*i.e.* the 7th argument is pushed *last* so that it is closest to the return address). The ***return value*** must be placed in `%rax`. For return values larger than a word size, we instead return a pointer to the return value in `%rax`.

Stack Frames

When we talk about "local procedure context," we naturally group execution by procedure calls. In order to enable recursion, we must support multiple simultaneous instantiations of individual procedures. To this end, we divide our stack into ***stack frames***, which hold the local state of each procedure instantiation.

Stack frames exist only for a limited time (from when the procedure is called until it returns). Any space allocated during the execution of the procedure must be deallocated in a parallel manner to ensure that `ret` correctly pops off the return address and completely deallocates the stack frame. This process of managing stack frames properly is known as ***stack discipline***.

It is important to note the consequences of the stack's last in-first out (LIFO) operation – the caller's stack frame cannot be deallocated until its callee's stack frame is. Old data in a deallocated stack frame should NOT be used because it is likely going to be overwritten by a different stack frame later!

In the example shown on the right, `main` first calls a function `foo`, which in turn calls a function `bar`. After `foo` returns, `main` then calls another function `baz`, which then overwrites parts of `foo`'s old stack frame (and possibly `bar`'s, depending on the frame size).