1 Topics

- Basic idea
- Sparc Specifics

2 Basic Idea

- Stack basics
  - Basic method invocation – stack grows and shrinks
  - But what is that stack for? Do we need it? For recursion we do.
  - Ok, assuming that we want it. What goes on the stack?
    - Locals
    - Parameters
    - Saved registers
    - Return address
    - Return value
- Parameters
  - Must include enough space to store all parameters
  - If these parameters are large (full structs or arrays), then must include enough space for the entire structure
- Locals
  - Will need space for all local variables
  - What about nested blocks?
    - Space for all nested declarations.
    - Can overlap in memory or keep separate.
    - Allocate all at beginning or as needed.
- Responsibilities
  - Who does what?
  - Caller vs. Callee saved registers.
  - Who allocates stack space?
    - Caller? Needs to know number of parameters and locals and size of spill space.
    - Callee? Where does the caller put the parameters?

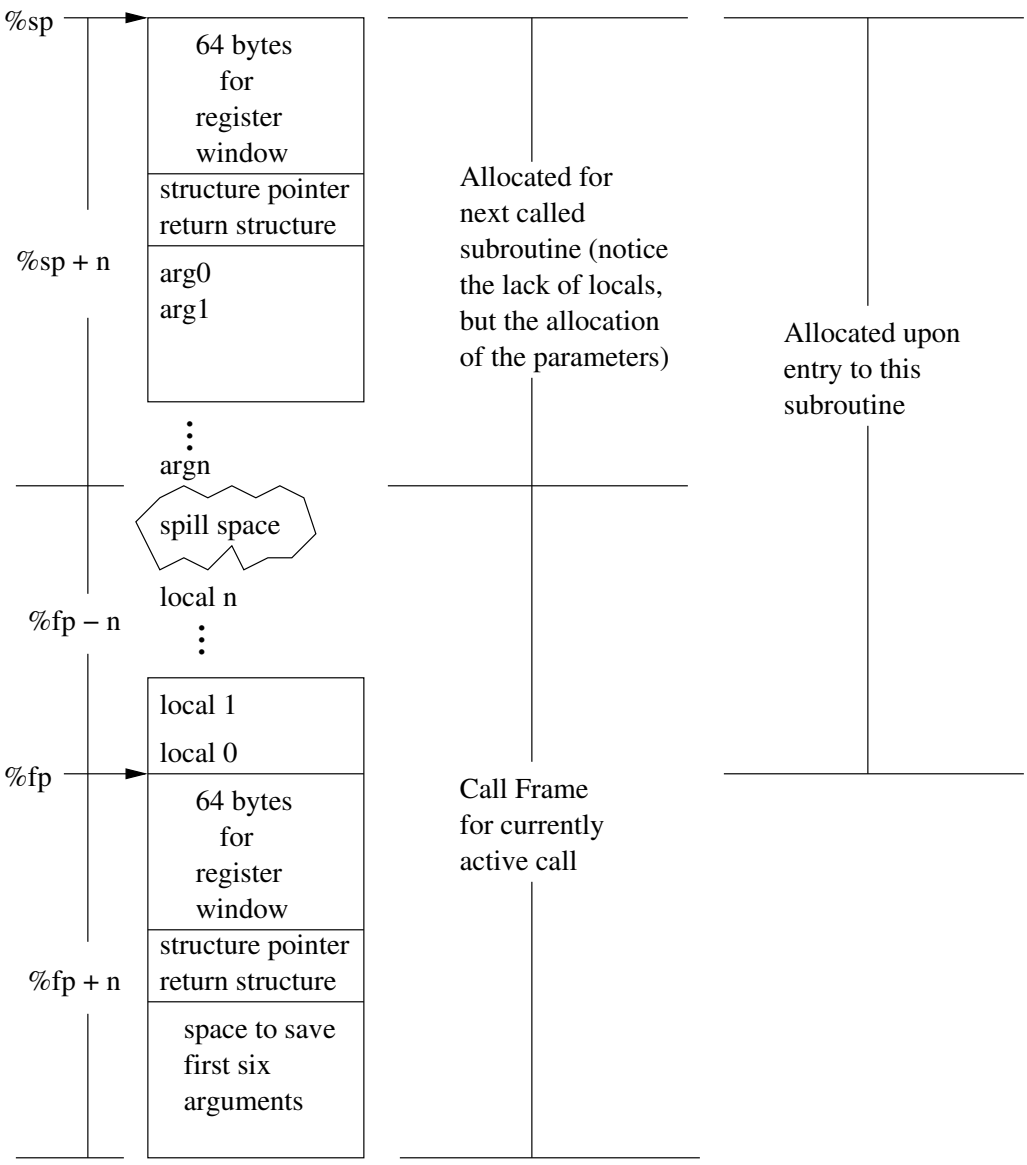
3 Sparc Specifics

- It turns out that the sparc is relatively simple in this regard. But with one important distinction.
- Registers
  - global — %g0 – %g7 — %g0 is always zero, these registers are not saved, they are global to all functions – %g6 and %g7 are reserved for the system in v9 – %g5 is also reserved in g8

  - local — %l0 – %l7 — for local values, automatically saved
  - out registers — %o0 – %o5 — first six actual arguments to a subroutine, can also be used as locals
  - in register — %i0 – %i5 — first six incoming formal arguments
  - special
    * %sp (o6) — stack pointer
    * %o7 — return address (stored here)
    * %fp (i6) — frame pointer
    * %i7 — return address
    * %i0 — put return value
    * %o0 — get returned value

- Register Window
  - All but the global registers are in a window.
  - These windows overlap. Specifically, the output registers of one window overlap the input registers of the next. (This includes %sp and %fp.)
  - The intent is that parameter passing will be quick.
  - Windows are changed via the `save` and `restore` instructions (and managed at a higher level by the OS).
  - What are the implications of this?
    * Not a whole lot.
    * First six arguments in registers.
    * More important once we get to register allocation.
    * Assembly programmer doesn't need to save and restore registers "by hand".

- Activation record
  - Odd thing. Upon entry, allocate space for *next* subroutine called.
  - Draw basic picture.



  - Align w/o locals and then align the entire thing
  - **IMPORTANT** But wait, if the current subroutine must allocate space (all but locals) for the next, then it must know how many arguments it takes.

- Arguments
  - Beyond the first six, arguments will be stored on the stack. But still required to provide space for the first six.
  - To access the arguments for this subroutine, add to %fp.

argument	offset
1	+68
2	+72
3	+76
...	
7	+92
8	+96
...	

- To access the arguments for a subroutine to be called, add to %sp.

argument	offset
1	+68
2	+72
3	+76
...	
7	+92
8	+96
...	

- Locals
  - Also accessed via the frame pointer (%fp).
  - This time an offset in the negative direction.

argument	offset
1	-4
2	-8
3	-12
...	

- GCC oddity
  - For some reason gcc pads the locals with 16 bytes at the beginning.
  - This changes the offsets to

argument	offset
1	-20
2	-24
3	-28
...	

  - This should have no effect on your code.
- Spill space
  - What's this spill space for?
  - Well, it's kind of like a local. But better.
  - Or it can be better. If one wants to save space (maybe).
  - Anyhow, there are a limited number of registers that can be used at any one time.
  - If one attempts to keep too many values in regsiters, then so will need to be spilled. They need to go somewhere.
  - If all locals are kept in registers (only if an address is never taken), then one can eliminate the need for the local space and just have spill space.

4 Example calls

- Multiplication

```
mov     9, %o0      ; operand one
mov     7, %o1      ; operand two
call    .mul
nop
mov     %o0, %l0    ; result
```

- malloc

```
mov     4, %o0     ; size to allocate
call    malloc
nop
mov     %o0, %l0   ; result
```