LECTURE 19

Subroutines and Parameter Passing

ABSTRACTION

Recall:

- *Abstraction* is the process by which we can hide larger or more complex code fragments behind a simple name.
- *Data abstraction*: hide data representation details.
- *Control abstraction*: create a well-defined operation by hiding execution details.

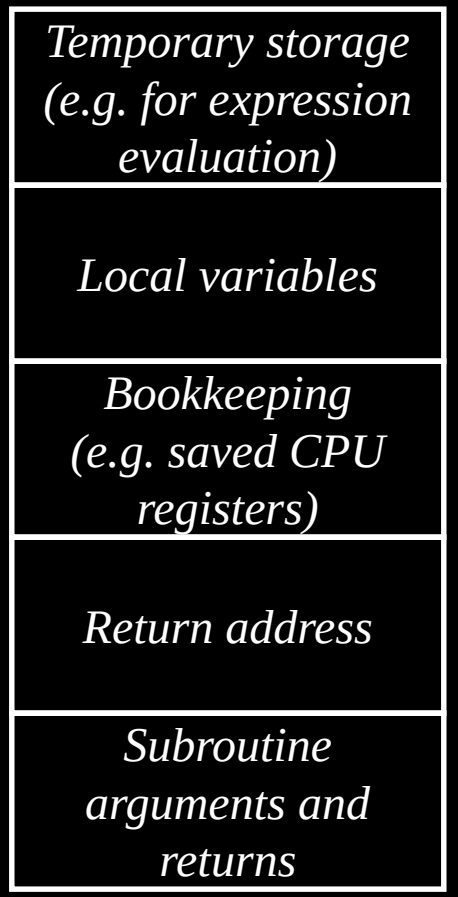
SUBROUTINES

Subroutines are the principle method of control abstraction in most languages.

- Subroutines are executed on behalf of a *caller* to whom the subroutine returns when it is finished.
- *Parameterized*: caller provides subroutine with arguments which influence execution of subroutine.
  - *Actual parameters*: actual arguments passed in a subroutine call instance.
  - *Formal parameters*: aliases for actual parameters within the subroutine.
- *Functions*: subroutines that return values (typically).
- *Procedures*: subroutines that do not return values.

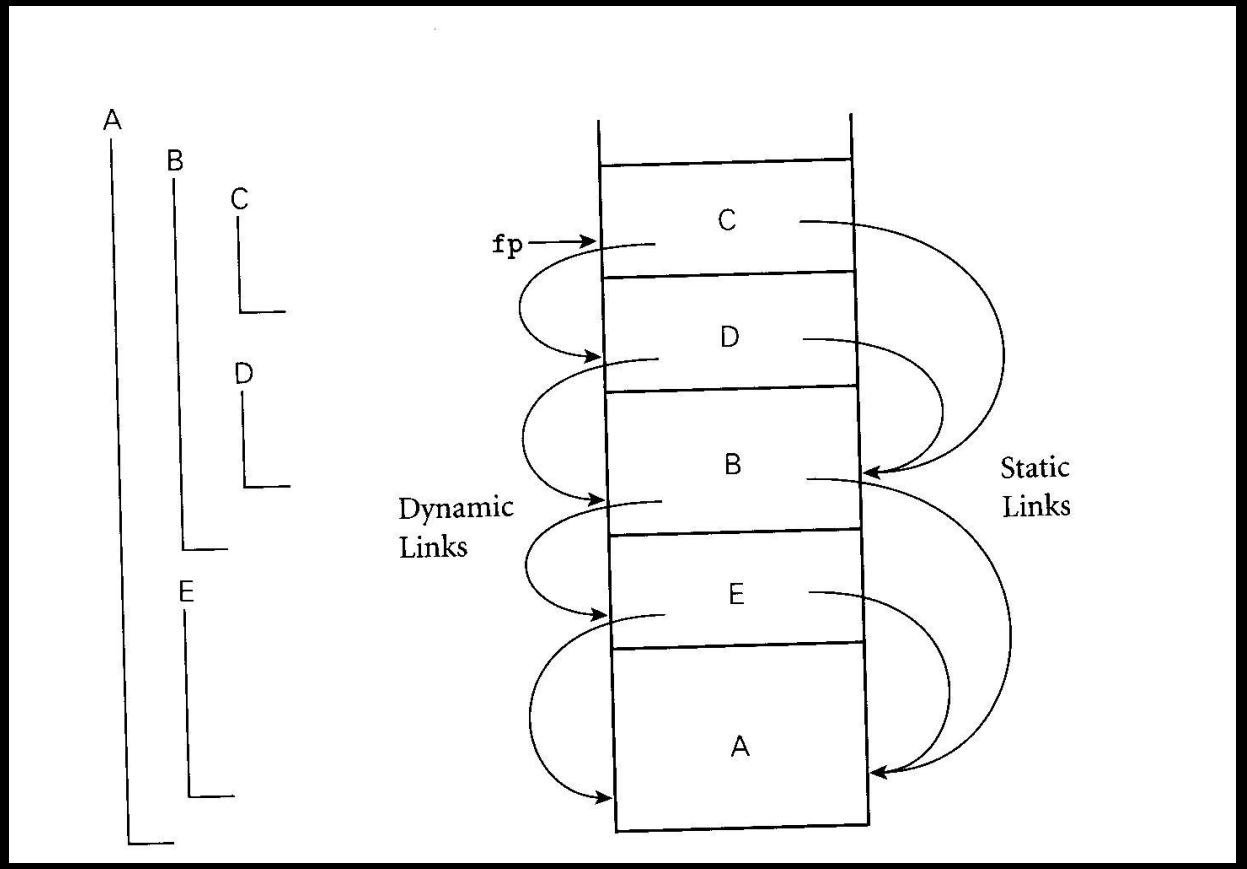
SUBROUTINE FRAME

- *An activation record (subroutine frame)* is used to store all related information for the execution of a subroutine.
- Before a subroutine is executed, the frame must be set up and some fields in the frame must be initialized.
- Formal arguments must be replaced with actual arguments.
- This is done in the *calling sequence*, a sequence of instructions before and after a subroutine, to set-up the frame.



SUBROUTINE FRAME

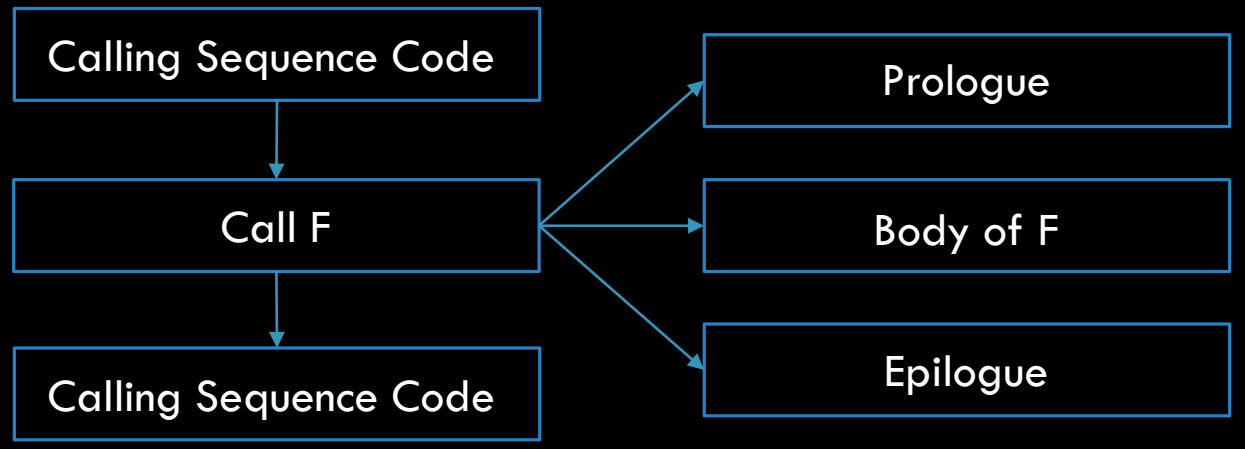
- With stack allocation, the subroutine frame for the current frame is on top of the stack.
- *Stack pointer (sp)*: top of the stack.
- *Frame pointer (fp)*: an address within the top frame.
- To access non-local variables, a static link (for static scoping) or a dynamic link (dynamic scoping) is maintained in the frame.



CALLING SEQUENCES

- Maintenance of the subroutine call stack is the responsibility of the *calling sequence*.
- In general, calling sequences have three components:
  - The code executed by the caller immediately before and after a subroutine call.
  - *Prologue*: code executed at the beginning of the subroutine itself.
  - *Epilogue*: code executed at the end of the subroutine itself.

```
…
F(a, b);
…
```



CALLING SEQUENCES

What needs to be done in the calling sequence?

Before the subroutine code can be executed, we need to set up the subroutine frame (calling code before the subroutine call + prologue). Generally speaking, the steps are:

- Compute the parameters and pass the parameters.
- Save the return address.
- Change the program counter.
- Change sp to allocate space for new frame.
- Save registers, including fp, which may be overwritten by subroutine.
- Change fp to allocate new frame.
- Execute initialization code when needed for objects in new frame, if needed.

CALLING SEQUENCES

What needs to be done in the calling sequence?

After the subroutine code is executed, remove the subroutine frame (calling code after the subroutine + epilogue). Generally speaking, the steps are:

- Pass return parameters or function value.
- Finalization code for local objects.
- Deallocate the stack frame (restore fp and sp to their previous values).
- Restore saved registers and program counter.

CALLING SEQUENCES

Some of the operations must be performed by the caller, while others can either be done by the caller or callee.

For example, passing parameters must be done by the caller because the parameters passed will differ every time a call is made.

In general, it's better to have set-up and tear-down operations performed by the callee as much as possible. This cuts down on code generated for subroutine calling as it only needs to appear once in the target program. On the other hand, any tasks that the caller is responsible for appear at every call site.

SAVING AND RESTORING REGISTERS

- Some registers such as sp and fp are clearly different in and out of a subroutine.
- For general purpose registers used in the caller and/or callee:

```
main(){                          foo(){
    …                                addi $s0,$0,0
    add $s0,$s1,$s2              }
    …
    call foo();
    …
    mult $s0,$s5
}
```

- To execute correctly, $s0 needs to be saved before foo() and restored after the subroutine returns.

SAVING AND RESTORING REGISTERS

- The compiler should generate code only to save and restore registers that matter.
  - If a subroutine does not use $s0, $s0 does not need to be saved in the calling sequence.
- Ideally, we should only save registers that are used in both the caller and the callee.
  - Difficult due to separate compilation: no information about callee when compiling caller, and vice versa.
- Simple solution (with unnecessary save/restore):
  - Option 1: caller saves/restores all registers it uses.
  - Option 2: callee saves/restores all registers it uses.
- Compromise: partition registers into two sets, one for the caller to save and one for the callee to save.

SAVING AND RESTORING REGISTERS

- MIPS and x86 processors implement the following compromise: non-special-purpose registers are divided into two sets of roughly equal size. One set is the caller's responsibility, the other set is the callee's responsibility.
- A callee assumes there is nothing of value in the caller-saved set.
- A caller assumes the callee will not destroy the contents of the callee-saved set.

The compiler will use callee-saved set for long-term values and will use the caller-saved set for transient values (unlikely to need to be saved across calls).

Result: caller-saved set is rarely saved by either party.

A TYPICAL CALLING SEQUENCE

Caller before the call:

1. Save any caller-saved registers whose values will be needed after the call.
2. Compute the values of arguments and move them into the stack or registers.
3. Compute the static link, and pass it as an extra hidden argument.
4. Use a special subroutine call instruction to jump to the subroutine, simultaneously passing the return address on the stack or in a register.

Prologue in the callee:

1. Allocate a frame (sp = sp – offset).
2. Save old fp on the the stack, update the fp.
3. Save callee-saved registers that may be overwritten in the routine.

Epilogue in the callee:

1. Move the return value into a register or a location on the stack.
2. Restore callee-saved registers.
3. Restore fp and sp.
4. Jump back to the return address.

Caller after the call:

1. Move the return value to wherever it is needed.
2. Restore caller-saved registers.

QUESTION

Why do local variables typically not have a default value (while globals do)?

```
int i;
int main(){
    int j;
    cout << i << j;
}
```

ANSWER

Global variables (and statics) are allocated a fixed section of memory at compile time so initializing them to zero incurs no real runtime overhead – it's a one-time deal when the program is loaded into memory.

Local variables, however, may have a different address on the stack every single time they are called which would require a new initialization step every single time space was made for them. It's easier to just leave it up to the programmer to explicitly request some value.

REGISTER WINDOWS

An example of hardware support for efficient subroutine execution is *Register Windows*.

- Introduced in Berkeley RISC machines.
- Also used in Sun SPARC and Intel Itanium processors.
- Maintain multiple sets (a window) of ISA registers using a much larger collection of physical registers.
- Use a new mapping (set) of registers when making subroutine calls.
- Setting and resetting a mapping is cheaper than saving and restoring registers.
- New and old mapping registers overlaps to allow parameter passing.

IN-LINE FUNCTIONS

An example of language support for efficient subroutine execution is *in-line functions*.

- In C/C++: `inline int max(int a, int b) {return a> b ? a: b;}`
- Such functions are not real functions, the routine body is expanded in-line at the point of call.
- A copy of the routine body becomes a part of the caller.
- No actual routine call occurs.
- Will inline function always improve performance? Maybe, maybe not.
- Many other factors: e.g. code size affecting cache/memory behavior.
- Most compilers decide on their own whether to use in-line or call conventionally.