Compilation 2014

Activation Records (Part 1)

Aslan Askarov
aslan@cs.au.dk

Revised from slides by E. Ernst

(Abstract) computer organization

- Program memory
  - code segment – contains program text
  - data segment – contains static program data (globals)

  } size fixed prior to runtime

  - stack – for locals and function arguments
  - heap – for dynamically allocated memory

  } size changes at runtime

- Processors registers
- Operations for moving data between registers/memory

Call stack of a program

Higher addresses in memory

contiguous region in memory that the program can use

the stack grows from higher memory addresses to low ones (could be otherwise depending on architecture)

Purpose of the stack:

- store local variables
- pass arguments
- store return values
- save registers
- …

Low addresses in memory

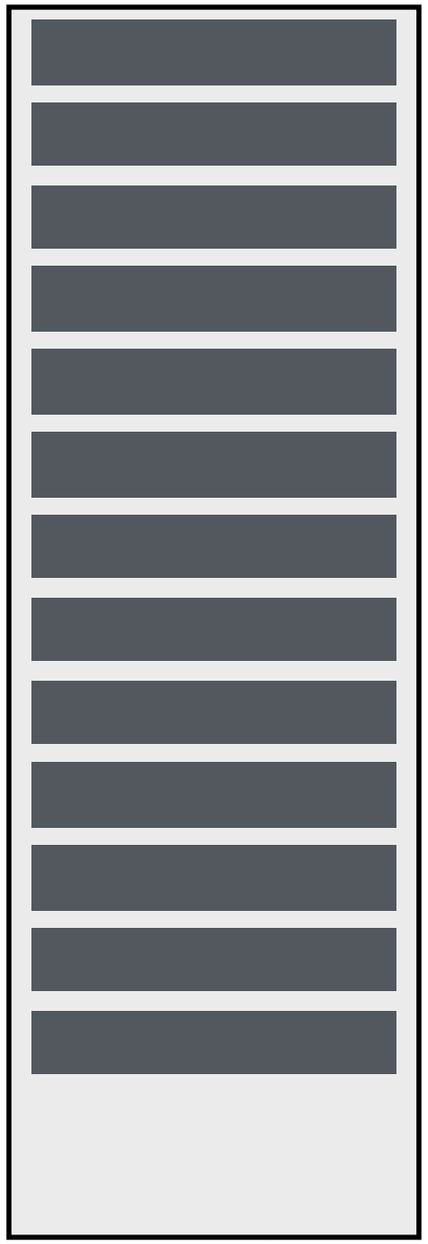
stack limit – set by OS

Call stack of a program

Higher addresses in memory

the stack grows from higher memory addresses to low ones (could be otherwise depending on architecture)

Low addresses in memory



contiguous region in memory that the program can use

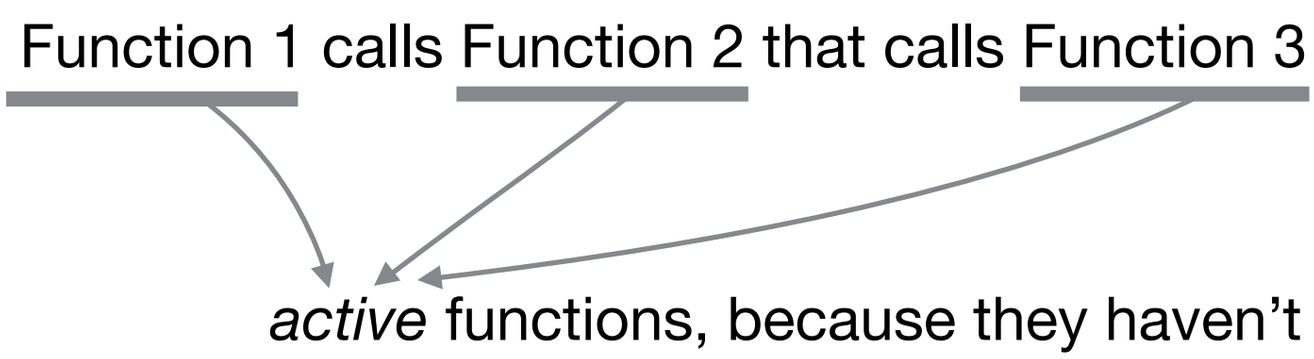
Purpose of the stack:

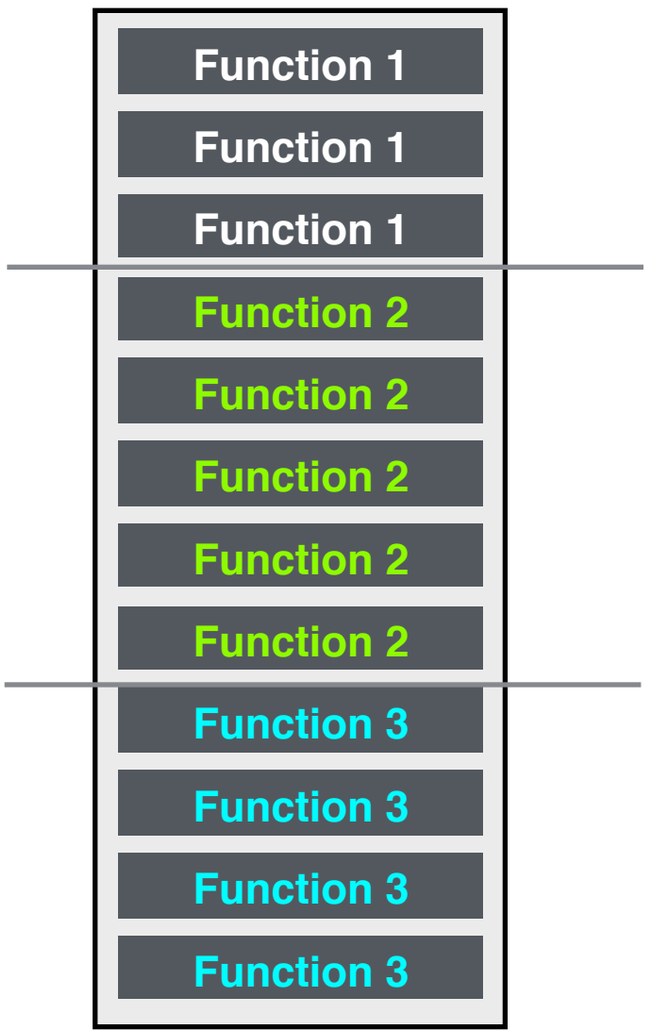
- store local variables
- pass arguments
- store return addresses
- save registers
- …

Q: *what happens if we push past beyond stack limit?*

stack limit – set by OS

Stack frames





stack manipulations by each of the functions

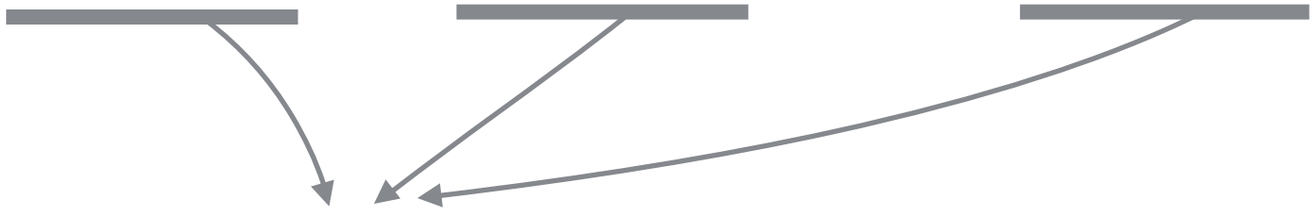
Idea: the maximum amount of memory that each function needs for its locals, temps, etc can be (usually) pre-computed by the compiler

Let's increase the stack by that much at once instead of many small increases

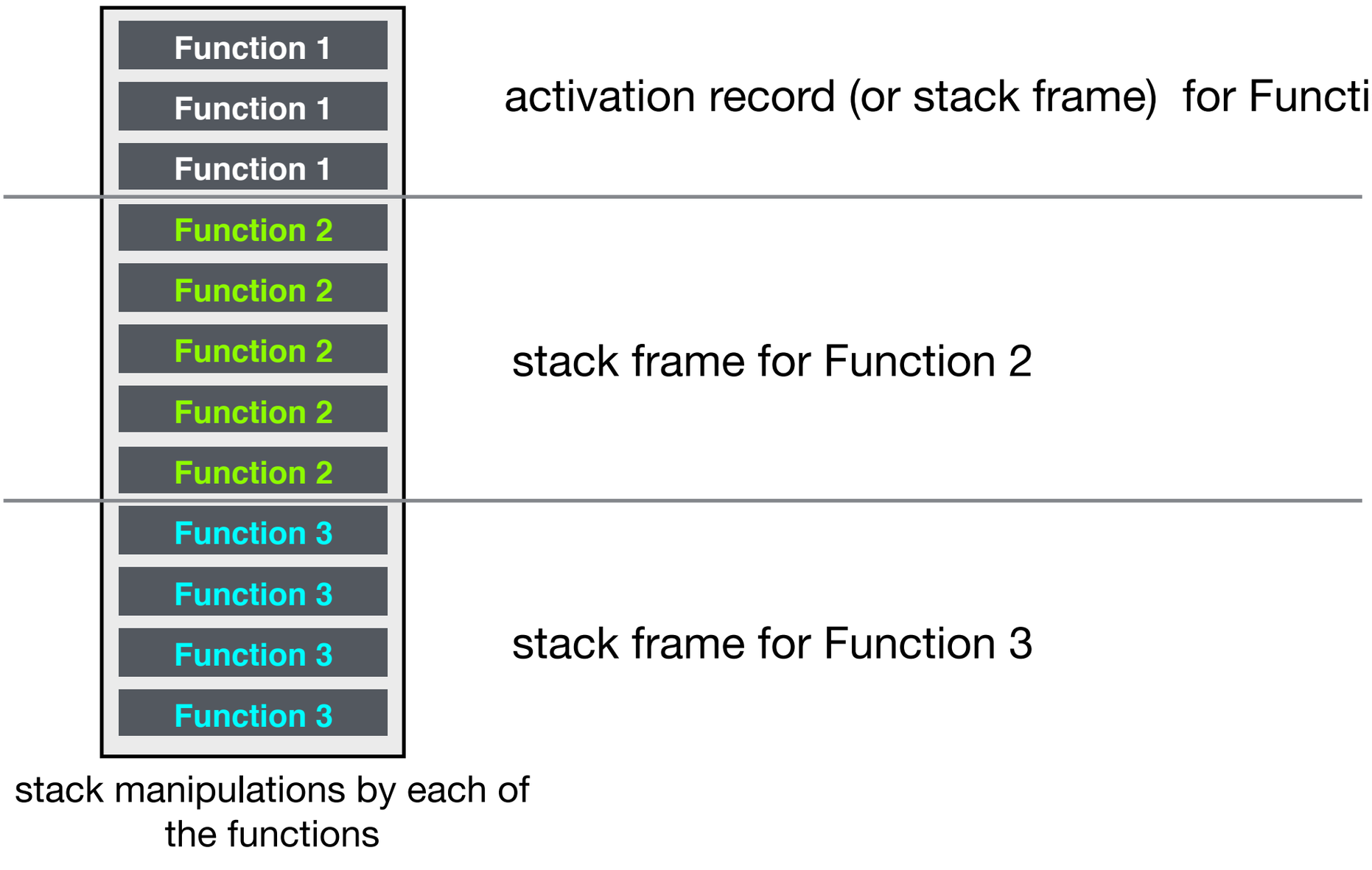
Call the region of the stack corresponding to each active function that function's *stack frame (*also called *activation record)*

Stack frames

Function 1 calls Function 2 that calls Function 3



active functions, because they haven't returned yet



activation record (or stack frame) for Function 1

stack frame for Function 2

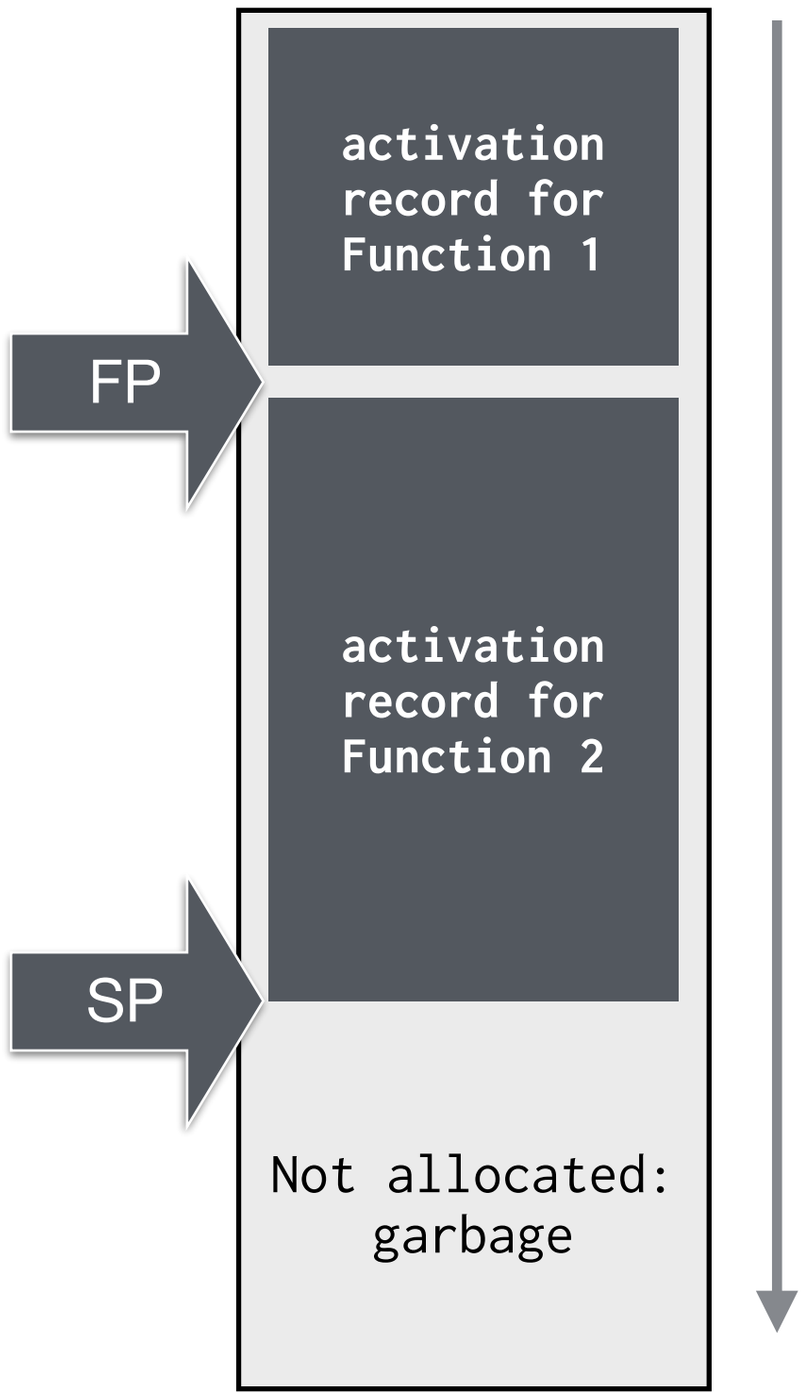
stack frame for Function 3

stack manipulations by each of the functions

Frame pointer and Stack pointer

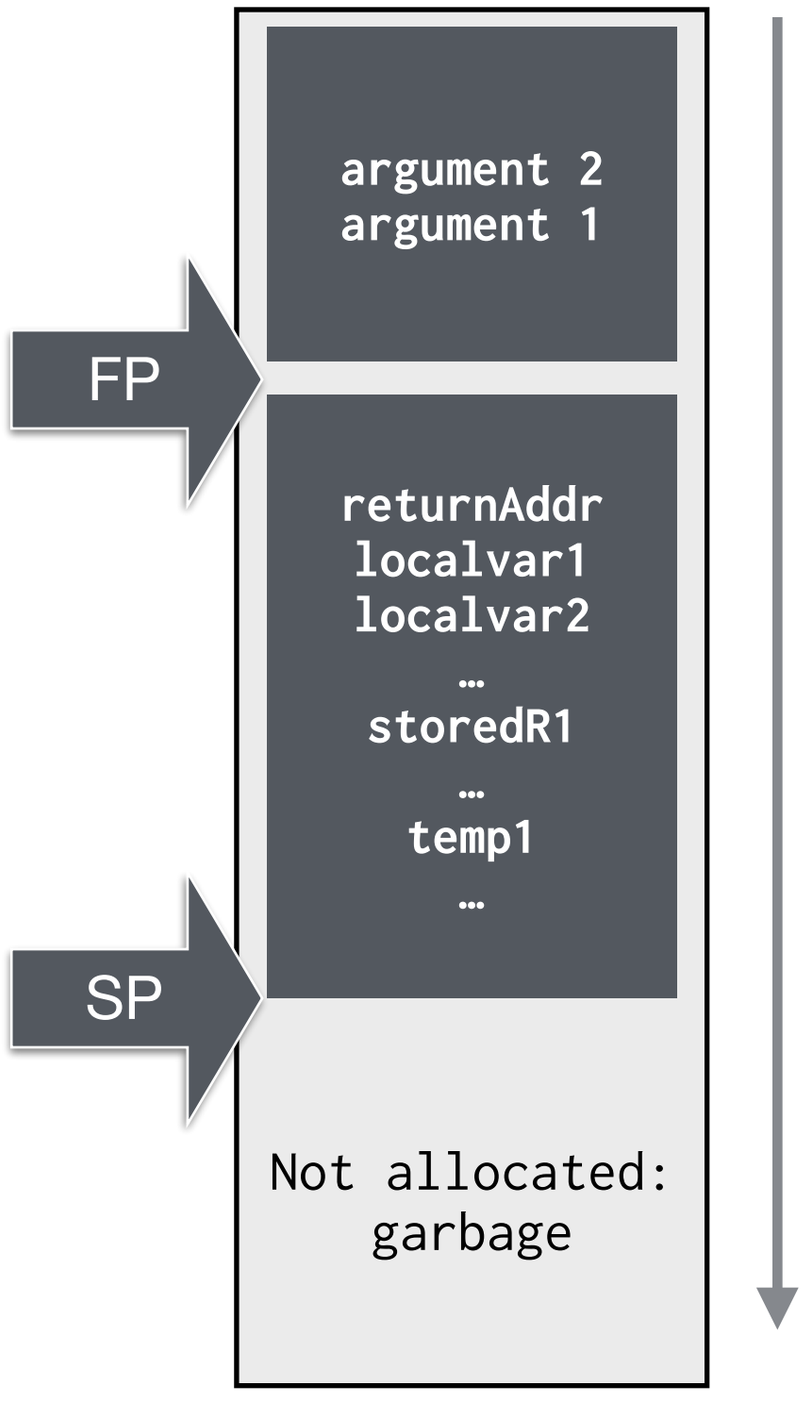
Stack pointer (SP): points to the “top” of the stack

Frame pointer (FP): the value of SP at the time the frame got activated



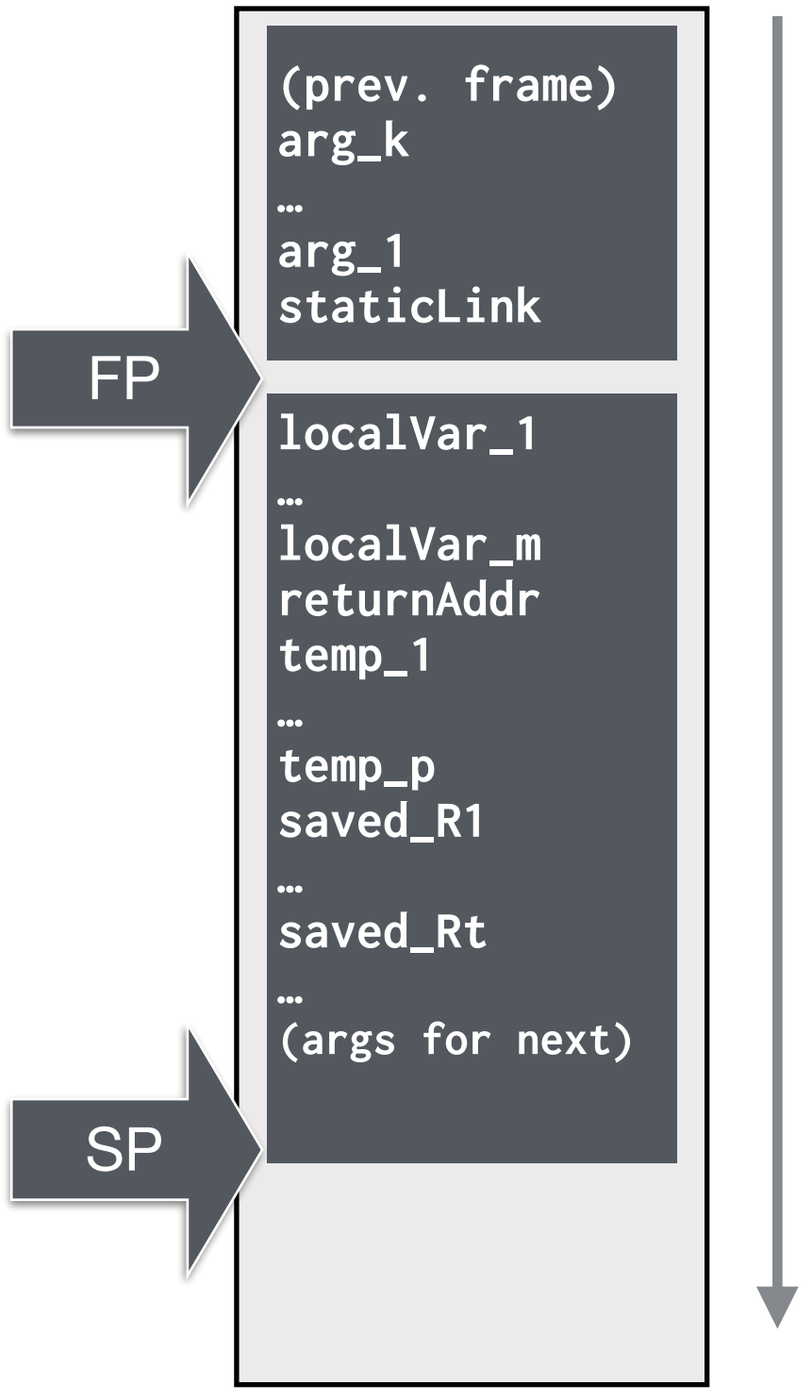
Frame layout: calling conventions

- Cross-language calls important: using libraries
- Reasonable to follow a standard: ‘calling convention’
- Specifies stack frame layout, register usage, routine entry, exit code
- Likely C bias



Typical frame layout

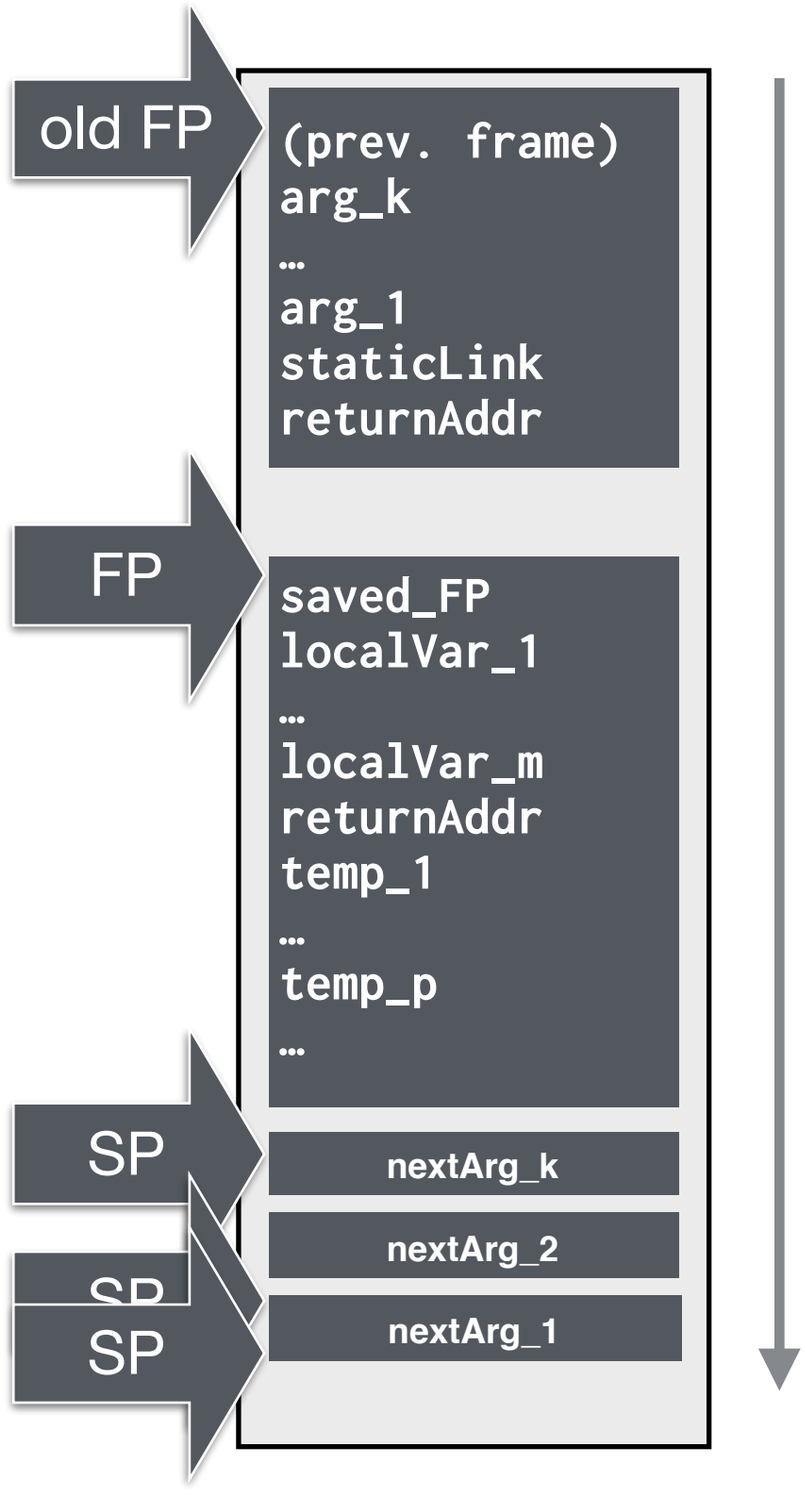
- Fits RISC architectures (such as MIPS) well
- Note ‘`staticLink`’...
- Consider offsets from FP and SP: are all known at compile time?
- FP could be virtual, if frame size is fixed



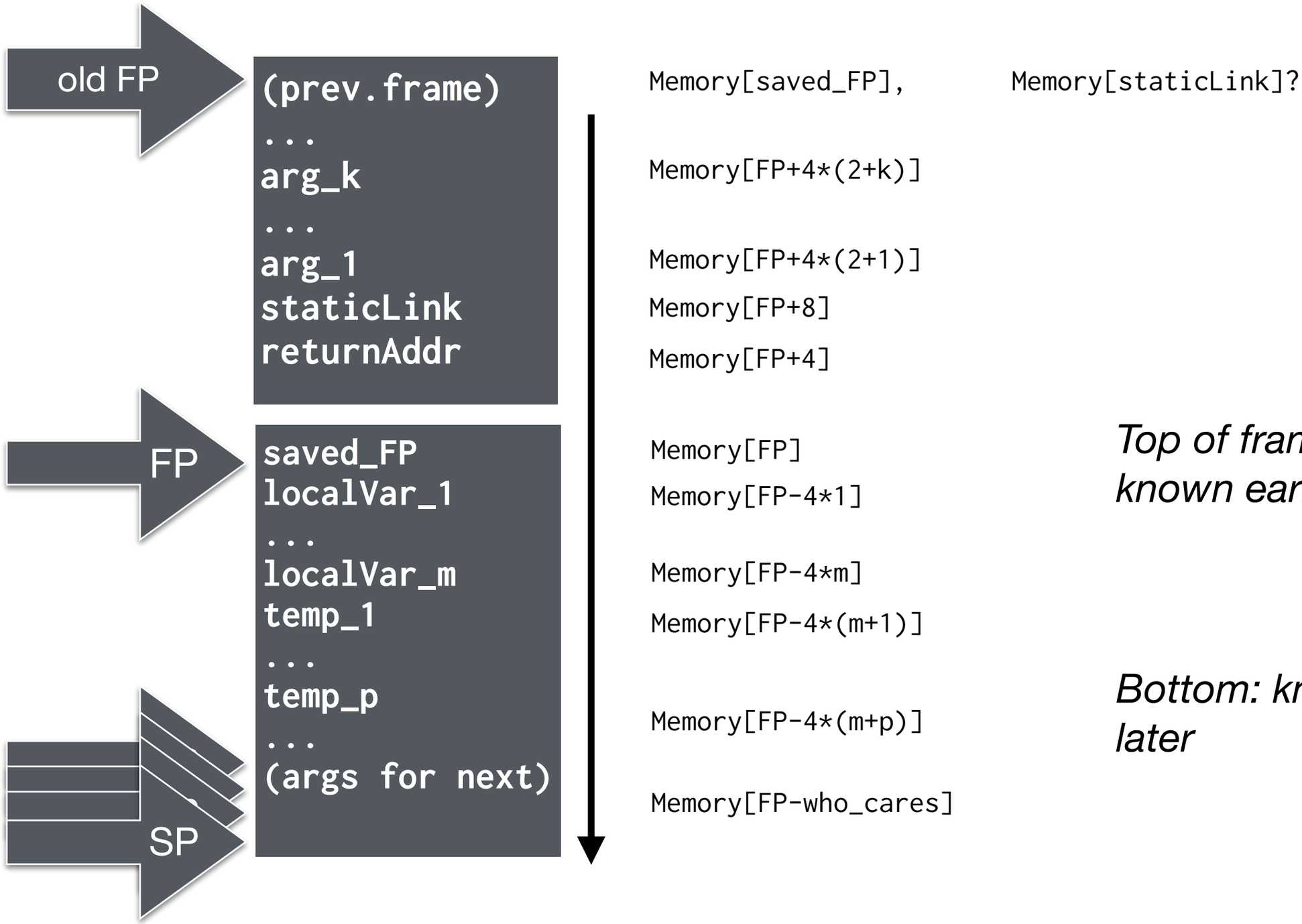
Our Tiger frame layout

- Fits x86, with simple register usage strategy

- Worth noting
  - return address pushed automatically by `call` instruction
  - FP non-virtual, always saved
  - SP adjusted at runtime: arguments pushed



Accessing Tiger frame slots

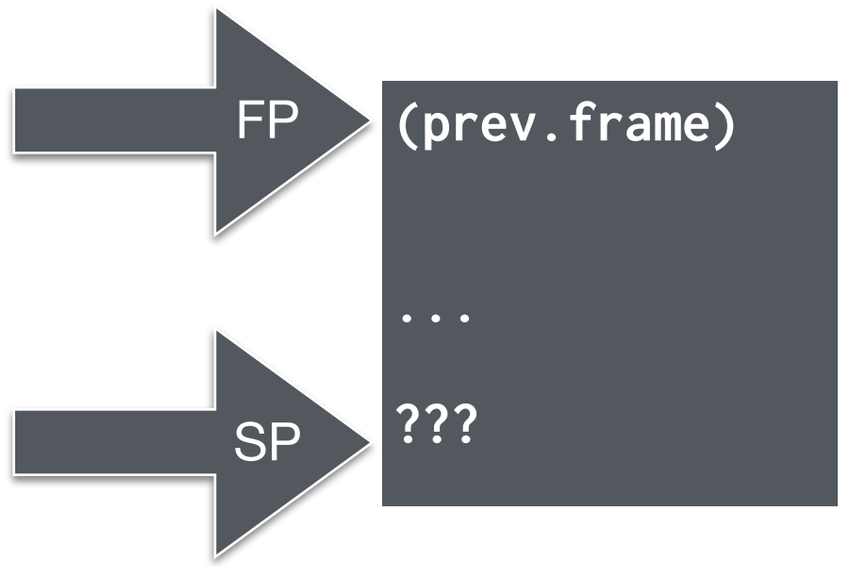


The Frame Pointer

- Relative concepts: caller/callee frames, routines

- On routine entry `SP` points to `arg_1` or `staticLink` or `returnAddr` ..
  (*will return to this later*)

- Allocate new frame (same figure):
  - `push FP`
  - `FP := SP`
  - `SP := SP - frameSize`



- If `SP` fixed, may replace `FP` by
  `SP - frameSize` (‘virtual’ FP)

Saving Registers

- Re: A frame/routine may be a caller or a callee

- Roles for registers: ‘Caller-save’ vs. ‘callee-save’
  - E.g. on MIPS: r16-r23 callee-save, others caller-save
  - “Don’t mess with r16-r23” / “Don’t rely on others”
  - Scenario: Nice if value in r3 is dead before call
  - Scenario: Put long-lived value in r21 (think: loop!)

Passing Parameters

- Pre-1960: Use globals, no recursion

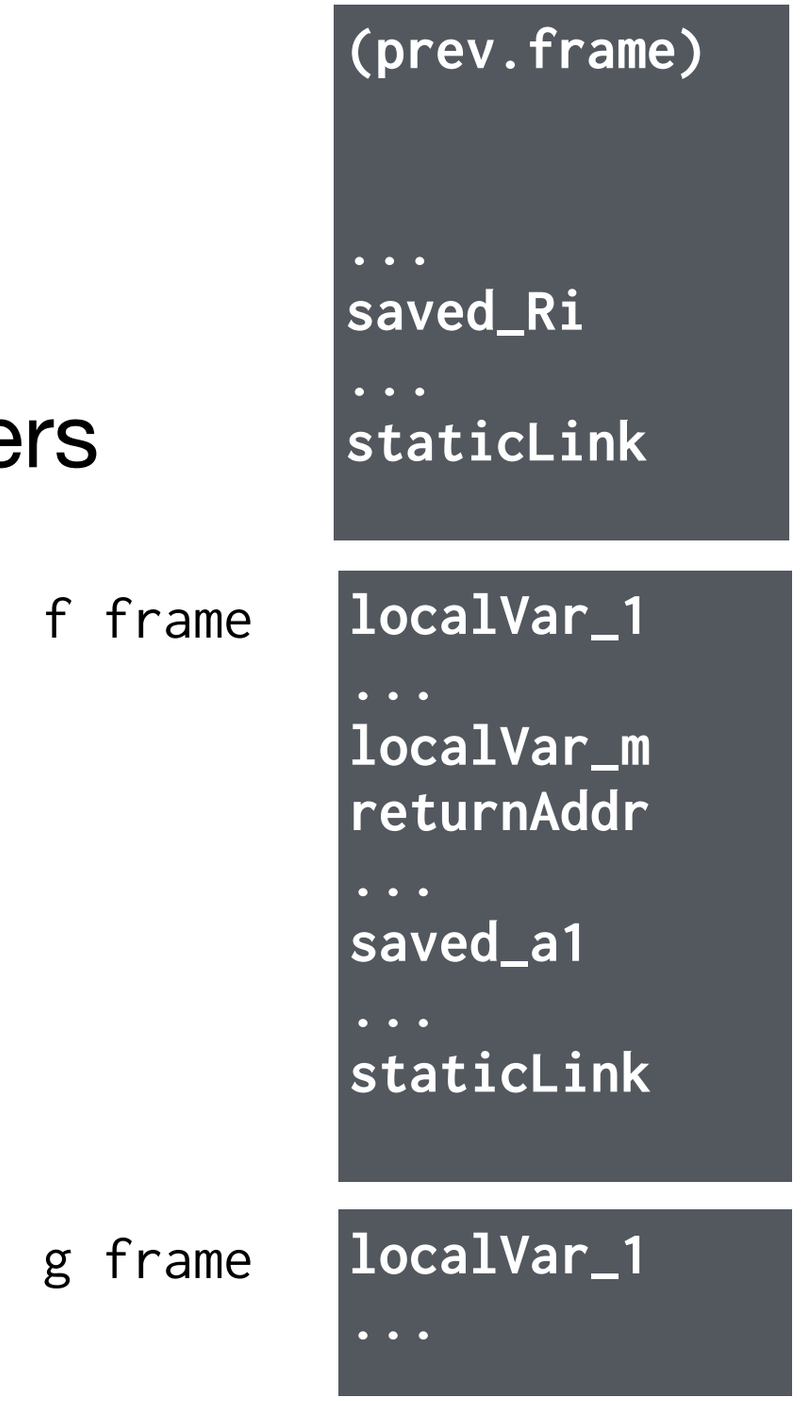
- 1970'ies: Pass parameters on stack
- Later: 4-6 parameters in registers, rest on stack

- Experience: Few routines >6 parameters!

- Our approach: Pass parameters on stack (fits x86, C calling conventions)

Why is register passing useful?!

- Scenario:
  - Calling f(a_1..a_n) which calls g(z)
  - As much as possible kept in registers
- Tempting claim: “No difference!”
- Concept: *Leaf routines*
- How rare?
  - Most invocations are leaves!
- May not even need stack frame



Some C Language Issues

- C extremely well-established, collaboration needed

- Address-of operator ‘&’ applicable to arguments
- ‘Varargs’ (printf) requires address arithmetics

- Allocate space for all arguments at end of frame, save only if address taken

- ‘Address taken’ also known as escaping, may use separate analysis

Managing Return Addresses

- Return address not statically known (.. why?)

- Solution: Store return address

- Old approach: Push at call instruction
- New: Store in register at call instruction

- Non-leaf routines will have to write to the stack

Forcing Memory Storage

- Using registers a good default, may fail...

- Address of variable taken (‘&’, pass-by-reference)
- Variable used from nested function

- Variable too large for register (use several?)

- Pointer arithmetics used (C arrays)

- Spilling