Data Segment and Linking

Computation Abstraction Stack	
Program Language	Java
Compiler	Eclipse, ...
OS/VM	JVM
Architecture / ISA	RISC-V x86
Microarchitecture	ALUs, FSM, Memory
Logic	AND
Transistors	
Electrons	

Source

- IA OS
- IA Java (OOP)
    - Object
- IB C and C++
    - `static`, `auto`, `extern`
- IB Computer Architecture
    - RISC-V Calling Convention
    - Application Binary Interface
- IB Compiler Construction
    - Compilers Principles, Techniques and Tools(2013)
        - Ch 7 Runtime Environments
    - 2014p3q4
- IB Concurrent System
    - Threads
- Adapted from Wiki Data segment

Symbol table (Linking)

```c
// Declare an external function
extern double bar(double x);

// Define a public function
double foo(int count)
{
    double sum = 0.0;

    // Sum all the values bar(1) to bar(count)
    for (int i = 1; i <= count; i++){
        sum += bar((double) i);
    }
    return sum;
}
```

Symbol name	Type	Scope
bar	function, double	extern
x	double	function parameter
foo	function, double	global
count	int	function parameter
sum	double	block local
i	int	for-loop statement

Memory

From lower to higher parts of memory (address space)

The stack has been separated into expression(code) and values (computation).

1. Code / Text segment

Hold program code and constants (string literals).

- read-only and fixed size

```c
char * message = "This is a string literal.";
```

	code	Instruction Fetch
0	INSTRUCTION	
1	INSTRUCTION	current instruction
2	INSTRUCTION	← code pointer (pc)
3	INSTRUCTION	

	code	Instruction Fetch
4	INSTRUCTION	

Note:

- Code is shared by all threads, shared libraries, and dynamically loaded modules in a process.
- per-thread program counters

2. Static Data

Global variables

- `static`
  - preventing variables or function from being called externally
- `extern`
  - symbol tables will `export` them
  - linker will resolve symbols (match `inputs` & `exports`)

Local `static` variables

- retain its value between function calls

Note:

- Global variables are shared by all threads, shared libraries, and dynamically loaded modules in a process.

2a. Initialized static data segment

```
(extern) int i = 3;
static int b = 2023;
void foo (void) {
    static int c = 2023;
}
```

2b. Uninitialized static data segment / Block Starting Symbol

Hold variables above and constants.

- that do not have explicit initialization in source code.
- will be initialized to zero in C by `exec`.

```
static int i;
static char a[12];
```

3. Heap

Hold dynamically allocated memory

- `malloc`, `calloc`, `realloc`, and `free`
    - which may use the `brk` and `sbrk` system calls to adjust its size (`mmap`/`munmap` to reserve/unreserve potentially non-contiguous regions of virtual memory into the process' virtual address space).
- For Java, all objects (`class`)

```
ptr = (int*)malloc(n * sizeof(int));
free (ptr);
```

For Virtual Machine (JVM)

- restrict stack elements to have fixed size
    - Put PAIR, CLOSURE on heap and place heap pointers on stack

Heap	Note
	Higher addresses
v2	
HT_PAIR	
v1	
HT_CLOSURE	
	Lower addresses

- begins at the end of the BSS segment and grows to higher addresses from there.
- The heap segment is shared by all threads, shared libraries, and dynamically loaded modules in a process.
- Garbage Collection enables the run-time system to detect useless data elements and reuse their storage.

4. Stack

Hold `auto` variables are also allocated on the stack.

- function parameters, local variables
    - when the procedure is called, space for its local variables are pushed onto a stack
    - when the procedure terminates, that space is popped off the stack

```
void f(int k){
    k++;
}

void main() {
    (auto) int j = 3;
    f(j);
}
```

activation record / stack frame, various types of linkage information

- the set of values pushed for
    - block entry and exit
    - procedure (function, method or subroutine) call and return
- **frame pointer** (fp) register %ebp
    - points to the base location *within* the current activation record
    - serves as an *anchor point*
        - fixed reference point to locate other elements of the frame
- dynamic/control link
    - points to the old frame pointer (the caller's frame)
    - when a block is exited or function returns (pop the callee from the stack)
        - set the stack pointer to be the value of the dynamic link of the current frame
    - forms dynamic chain (all frames linked together with dynamic links) .
- static/access link
    - point to the *most recent* stack frame of enclosing scope or null
        - access the non local free variables stored in previous activation records, for nested functions
        - there may be multiple frames, so point to the *most recent* one.
    - forms static chain (all frames linked from deeply nested to less nested with static links) .

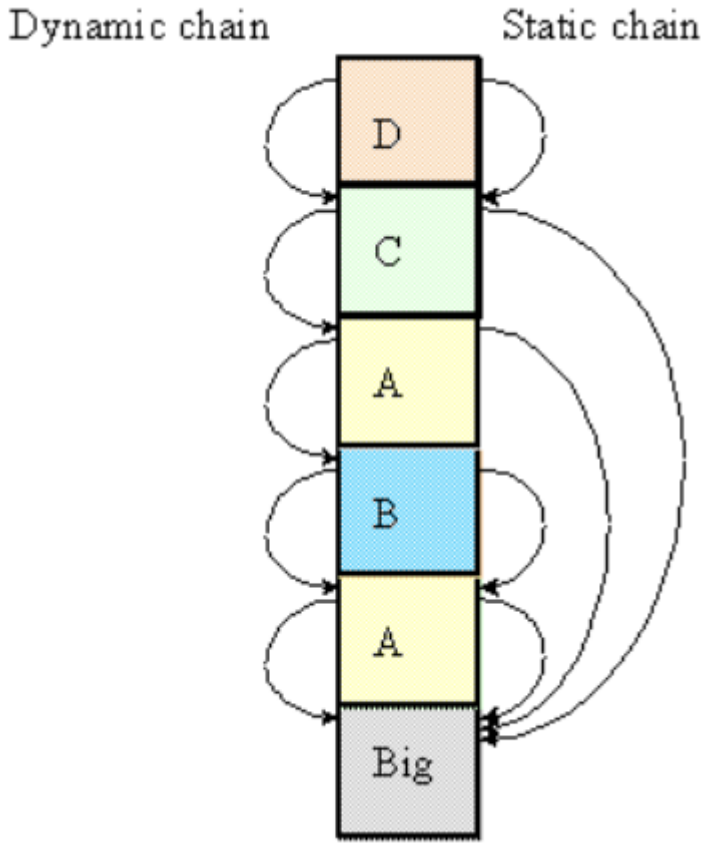
```
int i = 1;
void f(void){
    printf("%d",i); // free variable i
}
```

- **stack pointer** (sp) register %esp
    - points to the *top* of the stack
    - when a value is pushed onto the stack, adjust the stack pointer each time
    - when a block is exited or function returns, adjust the stack pointer to the dynamic/control link.
- for VM, store thread pointers (tp) / reference

Saver

- Caller-saved registers
    - volatile, call-preserved
    - a0-a7, t0-t6
- Callee-saved registers (bold below in RISC-V Column)
    - are non-volatile, call-clobbered
    - fp/s0, s1-s11, sp

Stack Frame	**RISC-V (ABI)**	**x86**
↓ code pointer (cp)	program counter (pc)	%eip
-- Previous Frame N-1 --	Higher addresses	
-- Current Frame N --	←**frame pointer (fp/s0)**	%ebp
dynamic/control link	point to old fp (frame N-1)	
return addr [must] saved registers	ra = caller pc / cp **data unchanged (s1-s11)**	%eip+1
access/static link	point to non-local/free variable at other stack frame	
	thread pointer (tp) →	
return values	optional (a0-a1)	
temporaries	local variables and not saved (t0-t6)	
parameters / arguments	optional (a0-a7)	
--- END of Stacks ---	←**stack pointer (sp)**	%esp
	Lower addresses	



Procedure and Stack

APPLY

- Caller
  - Push arguments onto the stack (in reverse)
  - Push the return address (ra)
    - the address of the instruction you want run after control returns to you
    - code pointer %eip+int
  - and Jump to the function's address

```
addi a0, a0, -1
CALL label = JAL ra, pc_offset
```

- Callee
  - Growing stack down (sp=sp-16)
  - Push the old frame pointer (fp/s0) onto the stack at dynamic/control link
  - Set frame pointer(fp) %ebp to the previous stack pointer(sp) %esp
  - Push local variables onto the stack; access them as offsets from %ebp

```
.label:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
```

RETURN

- Returning from function
    - Deallocate the stack frame (sp=sp+16)
        - %esp = %ebp
    - restore fp to the previous stack frame
        - pop ebp
    - Jump back to return address
        - pc = %eip

```
lw ra, 12(sp)
lw s0, 8(sp)
addi sp, sp, 16
ret = jalr zero, ra, 0
```

Note:

A callee should not make any assumptions about who is the caller because there may be many different functions that call the same function.

The frame layout in the stack should reflect this. When we execute a function, its frame is located on top of the stack. The function does not have any knowledge about what the previous frames contain.

- LIFO structure, growing towards the heap
- when the stack pointer met the heap pointer, free memory was exhausted.
- per-thread stack pointers and program counters

Memory Layout

Addr	Data Segment		ELF	Note
0x 8000	`int argc, char *argv[]`	command-line		
	Stack ↓	auto and linkage		
	----Free ---			
	Heap ↑	dynamic		
	Uninitialized static .bss	variables const	.data .rodata	initialized to 0 by `exec`
	Initialized static data	variables	.data	
0x 0000	Code / Text	program, const	.text	