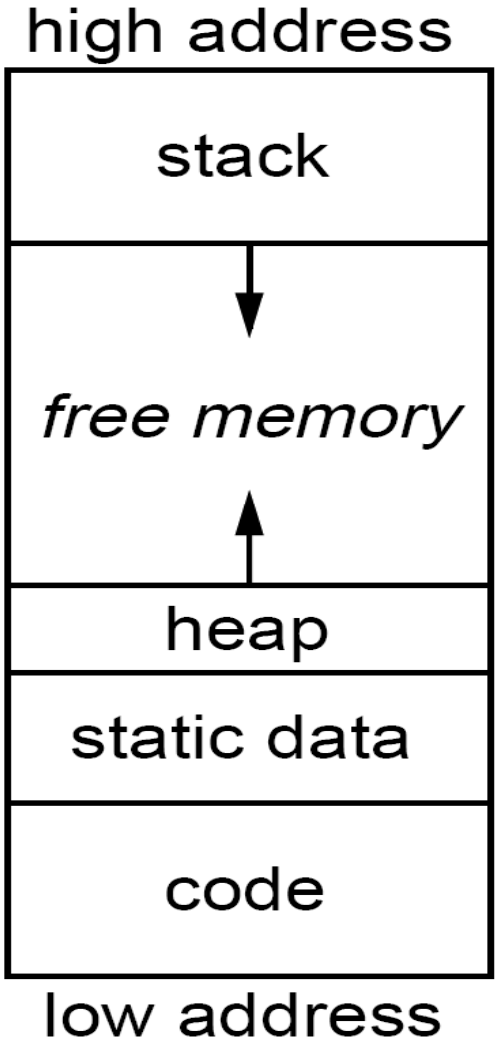
Chapter 6 Activation Records

Concepts

- Memory allocation methods for different kinds of variables.
- Using registers to store local variables and temporary results.
- Using registers to pass parameters and return results (for function calls).
- Stack frames (also known as activation records).
- Call/return sequence.
- Code injection attacks.

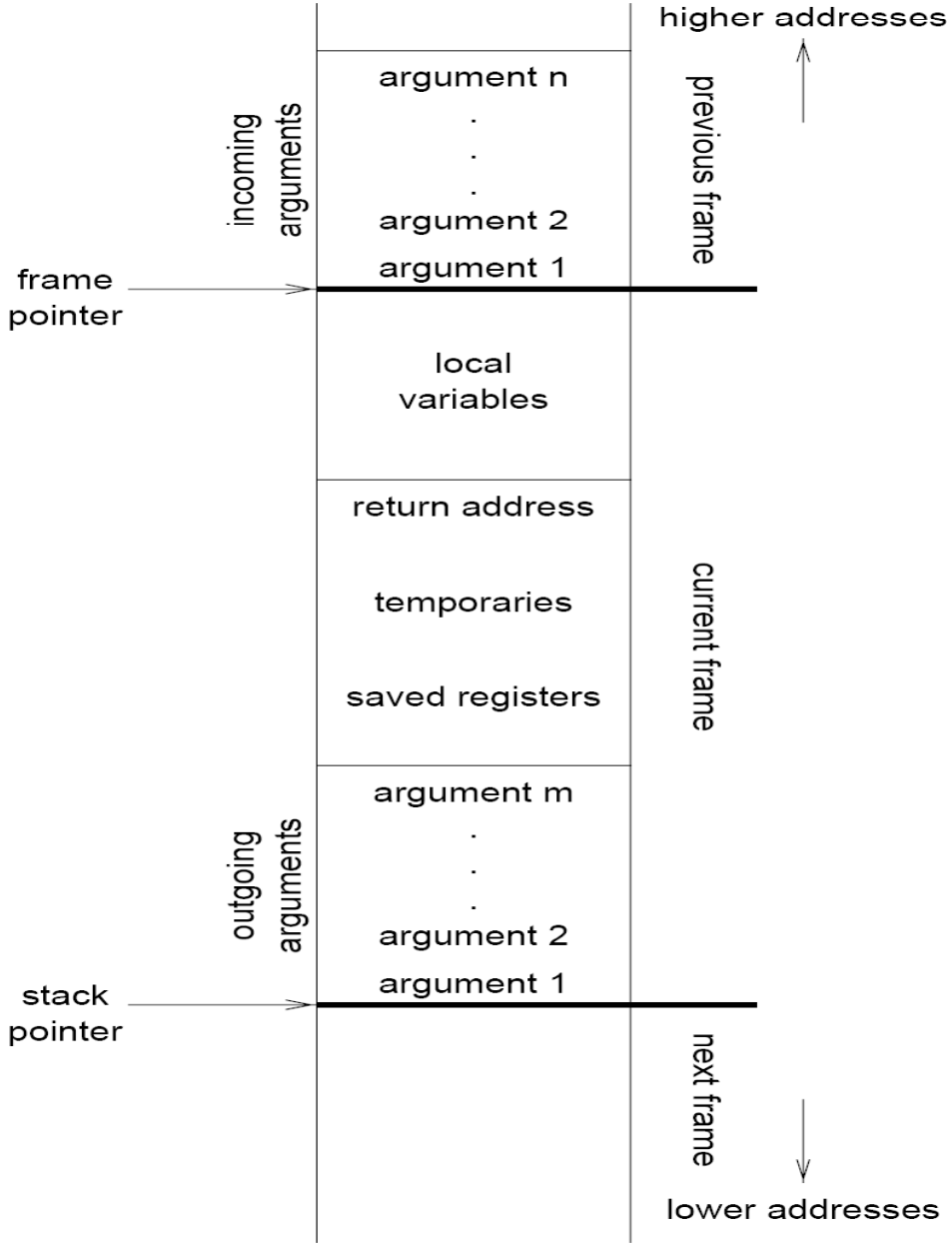
Runtime Image of Application Virtual Space

- A stack is maintained in the program's virtual address space. Variables local to a function are allocated to the stack frame, also known as the activation record, of that function.
- Variables and constants which are shared among different functions are allocated elsewhere.
  - Variables with fixed sizes known at compile time are allocated to static locations.
  - Dynamic data structures are allocated at run-time on the heap.



Stack Frame (Activation Record)

- Each procedure activation has an associated activation record or frame



Calling Sequence

- The following actions are divided between the caller and the callee:
  - 1. Evaluates actual arguments and puts values on the top of the caller's AR.
  - 2. Stores return address in caller's AR (sometimes in the callee's AR).
  - 3. Stores the caller's frame pointer register, or called the caller's AR pointer, in callee's AR. (Current AR pointer is called the control link in callee's AR.)
  - 4. Modifies the frame pointer %fp, making it point to callee's AR.
  - 5. Modifies the stack pointer %sp, making it point to the to top of the stack.
  - 6. Branches to callee's first instruction.
  - 7. Callee begins execution.
- Are there other register contents to be stored? Who stores them? Caller-save vs. callee-save.

Return Sequence

- 1. Caller needs to retrieve the function return value.
- 2. Restores saved stack pointer for caller (= current AR pointer).
- 3. Restores saved register contents for caller.
- 4. Return to the caller.

Demo One

- Use “gcc –g –o demo demo.c” to compile
- Use “objdump –d –S demo > dump” to disassemble the binary demo to dump
- Use “vim dump” to view the the disassembled code.

```
void foo (int x, int y) {
   int t;
   char name[16];
   t=7;
   if (x<0) return;
   foo(x-y,t);
}


int main( )
{
  foo(10,2);
}
~
```

Using Registers

- The memory references required to read and modify the stack contents can be time consuming. The number of such memory references can be reduced by using registers.
  - Passing parameters through registers.
    - Most functions have few parameters. We can use, e.g. two registers, Rx and Ry to pass parameters.
    - The rest of the parameters, if any, can be passed in the stack.
  - Returning function's results through registers.
- Dividing registers into two groups
  - Caller-saved registers.
  - Callee-saved registers.

Caller-Save or Callee-Save

- A leaf function (a function that makes no function calls) should use …
- Variables whose live ranges do not cover function calls should use …
- Suppose a variable's value is always dead (i.e. is no longer needed) in function ***g*** whenever ***g*** calls another function, say ***f***. Then that variable should use …
- If a variable is alive across multiple calls in ***f***, we should use …

Accessing Non-local Data

- Locals in outer procedures
  - Stack links (static links)
- Linked data structures (graphs, linked lists, variable length strings, ...) and other dynamically allocated data structures.
  - Heap

Code Injection Attacks

- The goal is to hijack a program execution.
- The idea is to overwrite the return address by overflowing a buffer in the frame.
- The consequence is that when the function returns, it returns to the malicious code.

Demo Two

```
#include "stdlib.h"

void foo(char * s) {
  int i;
  char c[4];
  int j;
  i=0;
  for (j=0;j<strlen(s);j++) {
    c[j]=s[j];
  }
  printf("i=%x\n",i);
}

int main () {
  foo ("aaaaaabb");
}
~
```

```
#include "stdlib.h"
int j;
void foo(char * s) {
  int i;
  char c[4];
  i=0;
  for (j=0;j<strlen(s);j++) {
    c[j]=s[j];
  }
  printf("i=%x\n",i);
}

//the pc of its entry is  0x08048421
void gee () {
  printf("I am in gee\n");
}

int main () {
  foo
("aaaaaabbaaaabbbb\x21\x84\x04\x08");
}
```

Execution result:

```
>./a.out
i=62626161
I am in gee
Segmentation fault (core dumped)
```

Given the following program:

```
int i, sum;

int SUM(int n) {
  i=0;
  sum=0;
  while (i<n) {
    i++;
    sum=sum+i;
  }
}
```

Given n=2, its execution trace is like

```
i=0;
sum=0;
while (i<n)
i++;
sum=sum+i;
while (i<n)
i++;
sum=sum+i;
while
```

How can you change the program so that executing the new program does the original computation and prints the trace. (Hint: recall how you use printf to print trace).

Please sketch a visitor pattern that automatically does the transformation for you. No class definition is needed, only brief discuss of your idea on visiting structures like CompositeStmt and WhileStmt, etc.