Discussion 7

CSE 131

address-of

- ptr = &x;
- x must be addressable, so we take the address of x and store it in ptr (as opposed to the value of x)

address-of

```
set  x, %l0
set  ptr, %l1
st   %l0, [%l1]
```

function pointers

- remember for normal function calling, you call with a label
  - a label is just an address
- function pointers build on that, except they use the addresses directly instead of the label
  - use call on a register instead of a label

function pointers

```
function : void foo() { /* */ }
function : void main() {
     funcptr : void() x;
     x = foo;
     x();
}
```

```
     .section ".text"
     .align   4
     .global  foo
foo:
     ...

     .global  main
main:
     set  SAVE.main, %g1
     save %sp, %g1, %sp
     set  foo, %l0
     st   %l0, [%fp-8]  ! tmp1
     ld   [%fp-8], %l0
     st   %l0, [%fp-4]  ! x in fp-4

     ld   [%fp-4], %l0  ! load x
     call %l0
     nop

     ret
     restore
SAVE.main = -(92 + 4 + 4) & -8
```

type casts

- depends on your work from project 1
  - in addition to compile-time conversion, now you have to handle run-time conversion
- remember a pointer is essentially an integer, so no value conversion between them necessary

static variables

- external (in global scope)
  - just like regular global variables, no .global
- internal (inside a function)
  - in data or bss, only initialized once

static variable example

```
int foo() {
     static int x = 5;
     return ++x;
}

int main() {
     cout << foo(); // 6
     cout << foo(); // 7
     cout << foo(); // 8
}
```

name mangling

- why? consider the following:

```
static int x;
void foo() {
     static int x;
     if (true) {
          static int x;
     }
}

void bar() {
     static int x;
}
```

extern variables

- what to do for variables declared "extern":
  - put on symbol table
  - access the variable using its name like any other variable
- the *only* assembly you generate is the instructions to access it
  - no space allocation, since they are declared elsewhere
  - the linker takes care of ensuring they exist

extern functions

- handled pretty much the same way
  - no label or body for extern function declarations
  - pretend the function exists when you call it
  - return types and argument types will only involve ints, bools, and pointer types
    - no float types passed directly

extern vs static vs global

module a	**module b**	**link-time error?**
`int x;`	`extern int x;`	no
`static int y;`	`extern int y;`	yes
`extern int z;`	`extern int z;`	yes
`extern int i;`	`int i;`	no

- note, it doesn't really matter which module above is rc.s and which is the separately linked object module

extern vs static vs global

```
int x;
static int y;
extern int z;

function : void main() {
     cout << x << y << z;
}
```

```
     .section ".bss"
     .align   4
     .global  x
x:   .skip    4
y:   .skip    4

     .section ".text"
     .align   4
     .global  main
main:
     set  SAVE.main, %g1
     save %sp, %g1, %sp

     ! print x
     set  _intFmt, %o0
     set  x, %l0
     ld   [%l0], %o1
     call printf
     nop
```

extern vs static vs global

```
    ! print static var y
    set  _intFmt, %o0
    set  y, %l0
    ld   [%l0], %o1
    call printf
    nop

    ! print extern var z
    set  _intFmt, %o0
    set  z, %l0
    ld   [%l0], %o1
    call printf
    nop

    ret
    restore
SAVE.main = -(92 + 0) & -8
```

runtime checks

- array out-of-bounds for array indexes that are unknown at compile time
- dereferencing or deleting pointers that are nullptr or calling function pointers that are nullptr

runtime array checks

```
int[3] myArr;
int x;
function : int main() {
     x = 47;
     x = myArr[x];
     return 0;
}
```

runtime array checks

- x = myArr[x]

```
    set  x, %l0
    ld   [%l0], %l0    ! value of x at runtime
    set  3, %l1        ! dimension of array (from declaration)
    cmp  %l0, %l1      ! checking upper bound
    bl   arrayLabel22
    nop

    /* do error message here and exit program */

arrayLabel22:
    /* continue checking lower bound (>= 0),
       then get value in myArr at offset x */
```

extra credit 1

- deallocated stack space is defined as the entire region of the stack that
  - starts at the lowest value of %sp that is reached during program execution
  - ends at the current value of %sp+92
- you will need to make memory address comparisons
  - use **unsigned** branches after the cmp instruction
    - bgu, bgeu, blu, bleu

extra credit 2

- consider creating a table or list of dynamic memory references in memory
  - keep track of every heap address returned by calloc and released by free

address	**attributes**
0x20003140	allocated
0x20004100	deallocated
...	...

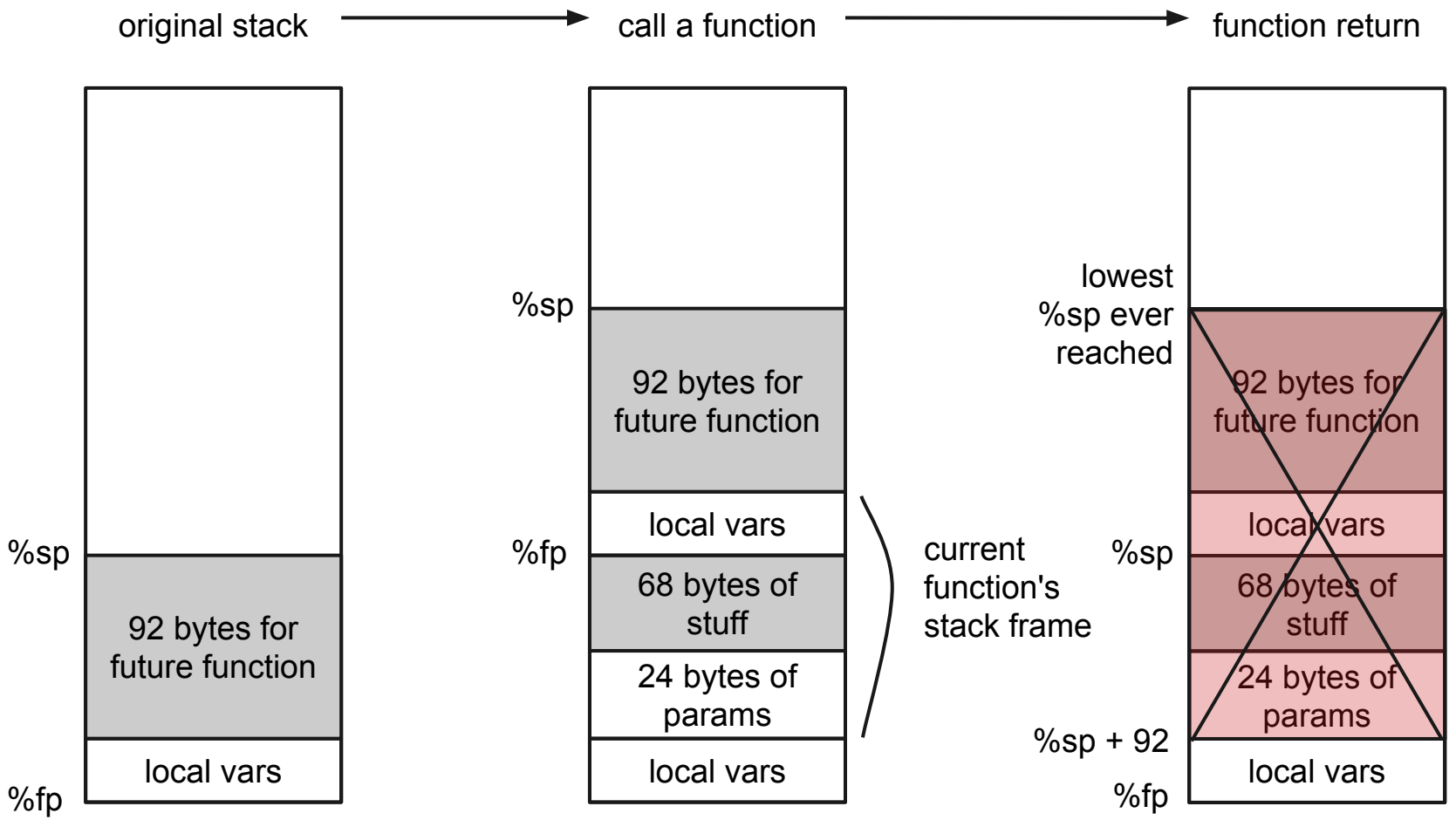
extra credit 2

- any attempts to delete deallocated space must be reported
- count the number of entries listed as "allocated" at the end of the program to report the number of memory leaks

extra credit 3

- passing more than 6 arguments to functions
  - see the example in the project 2 spec
- in a nutshell, caller needs to
  - allocate additional stack space
  - copy arguments into that space
  - call the function
  - deallocate that extra stack space

sparc stack layout



extra credit 4

- extend function overloading for code generation
  - if you did the extra credit in project 1, should be easy - just requires name mangling
  - we will not test assigning an overloaded function to a function pointer
  - we will not test overloaded functions declared as extern

extra credit 5

- passing structs by value
  - in caller
    - create a copy of each struct argument in a temporary and pass the copy of the argument by reference
  - in callee
    - access all struct parameters as though they were passed by reference, regardless of the "reference" flag

extra credit 5

- returning structs by value
  - remember the sparc stack layout
  - out of 92 bytes
    - 64 are space for %i0-%i7 and %l0-%l7 if they are evicted from the register bank, for example, during a context switch
    - 24 bytes for the first 6 arguments (param1 to param6)
    - 4 bytes for a struct pointer (%sp+64 in caller, %fp+64 in callee)

extra credit 5

- returning structs by value
  - in caller
    - allocate sizeof(struct) bytes of temporary space
    - store the address of that temp space in %sp+64
      - %sp becomes %fp in callee
  - in callee
    - copy contents of the return value into the address present in %fp+64 using memmove