CS 4400
Computer Systems

LECTURE 7

Representing procedure calls

New to C?: structs, unions, and functions

Procedure Calls

- A procedure call involves passing *data* (via procedure arguments and return value) and *control* from one part of the program to another.

- Each invocation of a procedure must allocate and deallocate memory in which to store its local variables.

Procedure Calls

- For IA32, very simple instructions transfer control:
  - `call, leave, ret`

- The compiler must generate additional instructions for passing arguments and allocation/deallocation of locals.

Run-Time Stack

- We use a *stack* as the LIFO data structure for holding local variable instantiations.
- The *stack pointer* is a special register (`%esp`) that always points to the "top" of the stack.

Run-Time Stack

- An abstract stack supports only *push* and *pop* operations.
  - However, local variables may be pushed (upon function entry) and popped (upon function exit) in large batches.
  - Also, after pushing on many variables, we may want to continue accessing variables deep in the stack.
  - Thus, we treat the stack as a large array.

Stack Frame

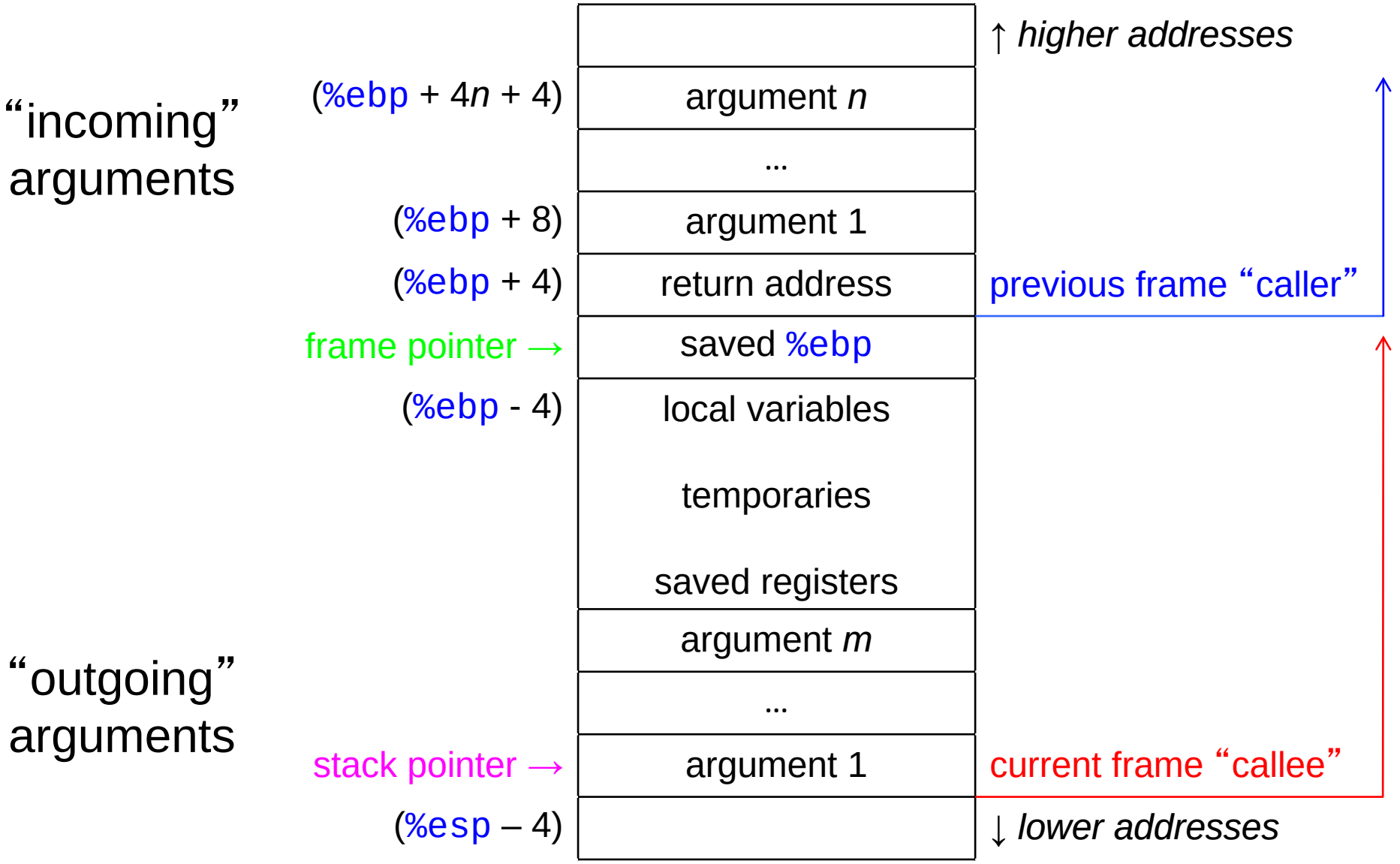
- A procedure's *stack frame* (or activation record) is the area on the stack devoted to its local variables, arguments, return address, and other temporaries.

- Usually, run-time stacks start at high memory addresses and grow to low memory addresses.
  - What addresses are "allocated"?
  - What addresses are "garbage"?

Stack Frame

- Often, each computer architecture has a standard stack frame layout, making it possible for procedures written in one language to call procedures written in another.

x86 Stack Frame Layout



More on Stack Frames

- Because the stack pointer can move while a procedure is executing, information is (most often) accessed using its address relative to the frame pointer.

More on Stack Frames

- When possible, local variables are stored in registers. Locals must reside in the stack when:
  - there are not enough registers (register spillage)
  - a local variable has its address taken (i.e. `int*p=&i`)
  - a local variable is an array or structure

- The return address is the address of the next instruction after the `call` instruction in the caller.

Transferring Control

`call` *label* and `call` *`*operand`*

- push the return address on the stack (`%eip` + 4)
- jump to the instruction indicated by *label* (or *operand*)

Transferring Control

`leave`

- prepare stack so that stack pointer points to return address
- equivalent to:

```
movl %ebp,%esp
popl %ebp
```

Transferring Control

`ret`

- pops return address from stack and jumps to that address

Register Usage

- All procedures must share a single set of registers.

- It is critical that the callee does not overwrite the contents of registers that the caller is still planning to use.
  - caller-save registers: `%eax, %edx, %ecx`
  - callee-save registers: `%ebx, %esi, %edi`

Register Usage

- caller-save registers: `%eax, %edx, %ecx`
- callee-save registers: `%ebx, %esi, %edi`

- Example:

```
int P(int x) {
  int y  = x * x;
  int z = Q(y);
  return y + z;
}
```

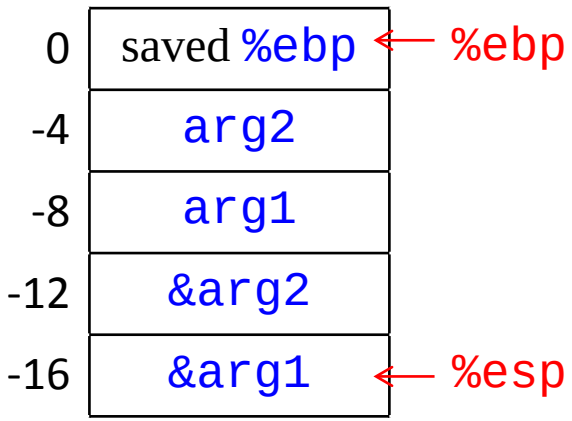
In what ways can `P` ensure that the value of `y` is available after `Q` returns? What is most efficient?

```
int swap_add(int *xp, int *yp) {
  int x = *xp;
  int y = *yp;

  *xp = y;
  *yp = x;
  return x + y;
}

int caller() {
  int arg1 = 534;
  int arg2 = 1057;
  int sum = swap_add(&arg1, &arg2);
  int diff = arg1 - arg2;

  return sum * diff;
}
```

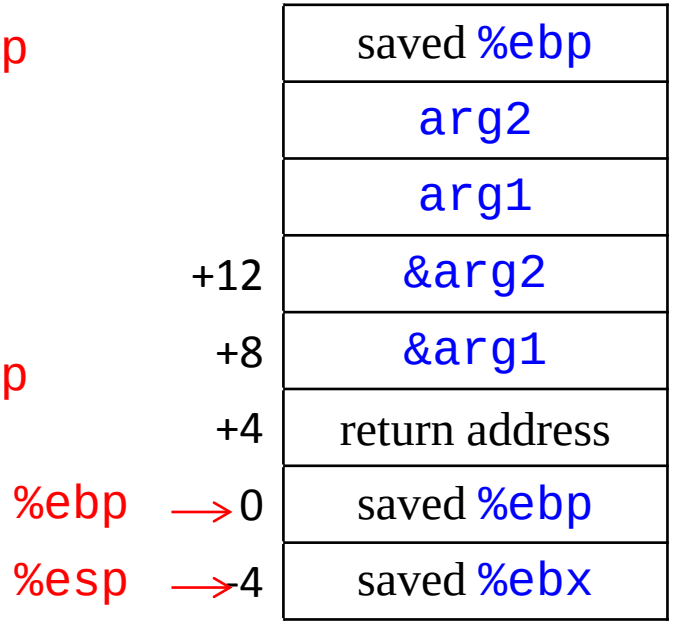
```
caller:
  ...
  leal -4(%ebp),%eax
  pushl %eax
  leal -8(%ebp),%eax
  pushl %eax
  call swap_add
  movl %eax,%edx
  ...
swap_add:
  pushl %ebp
  movl %esp,%ebp
  pushl %ebx
  movl 8(%ebp),%edx
  movl 12(%ebp),%ecx
  movl (%edx),%ebx
  movl (%ecx),%eax
  movl %eax,(%edx)
  movl %ebx,(%ecx)
  addl %ebx,%eax
  popl %ebx
  movl %ebp,%esp
  popl %ebp
  ret
```

prologue

epilogue



just before call



in body of swap_add

Exercise: Procedures

```
int proc(void) {
  int x, y;
  scanf("%x %x", &y, &x);
  return x-y;
}
```

- Where are the locals stored?
- How does run-time stack look?

```
_proc:
  pushl %ebp
  movl  %esp, %ebp
  subl  $24, %esp
  call  L1$pb
L1$pb:
  popl  %eax
  leal  -4(%ebp), %ecx
  movl  %ecx, 8(%esp)
  leal  -8(%ebp), %ecx
  movl  %ecx, 4(%esp)
  leal  L_.str-L1$pb(%eax), %eax
  movl  %eax, (%esp)
  call  _scanf
  movl  -4(%ebp), %eax
  subl  -8(%ebp), %eax
  addl  $24, %esp
  popl  %ebp
  ret
```

New to C?: Structures

- In C, a user-defined type is accomplished with a `struct`.
- Example:

```
struct element {
  char name[10];
  float mass;
};
```

- The new type is `struct element`.
- Declaration of a structure variable

```
struct element e1;
```

  - allocates contiguous storage for all structure members.
  - Size: (10 + 5 + 2 * `sizeof(float)` bytes)

More on Structures

- To access a member of the structure variable, use the dot . operator

```
e1.mass = 3.0;
strcpy(e1.name, “hydrogen”);
```

- Use typedef to avoid the awkward two-word type.

```
typedef struct element {
        char name[10];
        float mass;
} ELT;

ELT e1;
```

- What is the difference in a structure and an array?

Pointers to Structures

- As with objects in C++, the pointer operator -> can be used with pointers to structures.

```
ELT e1;
ELT* elt_ptr = &e1;
printf("%s", (*elt_ptr).symbol);
printf("%s", elt_ptr->symbol);
```

- A self-referential structure declaration has a member that is a pointer to an instance of itself.

```
typedef struct node {
   int data;
   struct node* next;
} NODE;
... x->next->next->data ...
```

Bitfields

- Bitfields let us create struct members that have non-standard bit-widths

```
struct pixel {
   int r:4;
   int g:4;
   int b:4;
};
```

- Bitfields are accessed just like regular struct fields using the . and -> operators

- A bitfield can be smaller than its type but not larger
  - Minimum size is 1 bit

Bitfields

- By default, bitfields are packed into words
- Compiler generates shifting and masking code necessary to access bitfields
- When should you use bitfields?

Bitfields

- By default, bitfields are packed into words
- Compiler generates shifting and masking code necessary to access bitfields
- When should you use bitfields?
  - If you have a very large number of struct instances, they can save memory
  - Sometimes they are a convenient way to access hardware registers when writing a device driver

New to C?: Unions

- Unions provide a way for a single object to be referenced according to multiple types.

- Example:

```
union u {
   char c;
   int i[2];
   double v;
} x;
x.v = 4.5;
printf("%d %d\n", x.i[0], x.i[1]);
```

- `sizeof(union u)` is the max size of any of its fields.

- Technically, you should only read the variant you wrote.

New to C?: Dynamic Memory Alloc

- For allocation of memory at run time, library routine `malloc` is used.
  - arguments specify number of bytes to be allocated
  - return value is a pointer to the allocated memory or `NULL`

- `malloc` allocates one contiguous block (of specified size).

```
NODE* head = malloc(sizeof(NODE));  // implicit
head->next = malloc(sizeof(NODE));  // cast
```

New to C?: Dynamic Memory Alloc

- To release dynamically-allocated memory, the library routine `free` is used.
  - argument is the pointer to the block of memory to be released

```
free(ptr);
```

New to C?: Parameter Passing

- In C, parameters are passed by value.
  - get the effect of call-by-reference by passing an address
- Array names are pointer constants.
- For a structure variable argument, its value is its content.
  - Unlike Java, where a declaration `ELT e` means that the value of `e` is a reference to an `ELT` object
- Which parameters may be modified from caller's view?

```
foo(char a, int b[], ELT c, float* d, NODE* e)
```

New to C?: Function Pointers

- Like an array name, a function name is a pointer constant.
- Why have function pointers?

```
void sort(int (*fn)(int, int), int arr[], int size) { ... }
```

New to C?: Function Pointers

- Like an array name, a function name is a pointer constant.
- Why have function pointers?
  - We can pass a function as an argument to another function.

```
void sort(int (*fn)(int, int), int arr[], int size) { ... }
int compare_incr(int a, int b) { return a < b; }
int compare_decr(int a, int b) { return a > b; }

int main(int argc, char* argv[]) {
  int a[8] = {5, -8, 19, 0, 2, 11, -90, 34};
  if(strcmp(argv[1], “ascending_order”) == 0 )
    sort(compare_incr, a, 8);
  else
    sort(compare_decr, a, 8);
  return 0;
}
```