The machine uses the stack to:
- Pass procedure arguments
- Store return information
- Save registers for later restoration
- Local storage

Stack frame:
- Portion of the stack allocated for a single procedure call
- The topmost stack frame is delimited by two pointers:
  - Register %ebp – the frame/base pointer
  - Register %esp – the stack pointer
    - Can move while the procedure is executing HENCE
    - MOST INFORMATION IS ACCESSED RELATIVE TO THE FRAME/BASE POINTER
    - Indicates lowest stack address i.e. address of top element
Procedure P (the “caller”) calls procedure Q (the “callee”)

**Caller stack frame (P)**
- The arguments to Q are contained within the stack frame for P
- The first argument is always positioned at offset 8 relative to %ebp
- Remaining arguments stored in successive bytes (typically 4 bytes each but not always)... +4+4n is return address plus 4 bytes for each argument.
- When P calls Q, the return address within P where the program should resume execution when it returns from Q is pushed on to the stack

**Callee stack frame (Q)**
- Saved value of the frame pointer
- Copies of other saved registers
- Local variables that cannot all be stored in registers (see next slide)
- Stores arguments to any procedures it calls.

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

- You can allocate space on the stack by decrementing the stack pointer by an appropriate amount (push).
- Space can be deallocated by incrementing the stack pointer (pop).
- Local variables on the stack:
  - There are a limited number of registers.
  - Stack is used to store local variables some of which can be arrays or structures.
  - When using & applied to a local variable, there must be an address than can be generated for it.
Procedure call and return

- **Call instruction**
  - Has a label which is a target indicating the address of the instruction where the called procedure (the callee) starts
  - Direct or indirect label
  - Push a return address on the stack
    - Is the address of the instruction immediately following the call in the (assembly) program
  - Jump to the start of the called procedure

- **Return instruction**
  - Pops an address off the stack
  - Jumps to this location
  - FYI: proper use is to have prepared the stack so that the stack pointer points to the place where the preceding call instruction stored its return address

- **Leave instruction is equivalent to:**
  - movl %ebp, %esp
  - popl %ebp
Procedure call and return

// Beginning of function sum
08048394  <sum>:
  8048394:  55  push %ebp
  ...
//return from function sum
  80493a4:  c3  ret
  ...
// call to sum from main - START HERE!
  80483dc:  e8 b3 ff ff ff  call  8048394  <sum>
  80483e1:  83 c4 14  add $0x14,%esp

Return address
Register usage conventions

- Program registers are a shared resource
- One procedure is active at a given time
- Don’t want the callee to overwrite a value the caller planned to use later

**BY CONVENTION/PROTOCOL**

- “Caller-save” registers: `%eax`, `%edx` and `%ecx
  - When Q is called by P, it can overwrite these registers without destroying any data required by P

- “Callee-save” registers: `%ebx`, `%esi` and `%edi
  - Q must save these values on the stack before overwriting them, and restore them before returning

- `%ebp` and `%esp` must be maintained

- Register `%eax` is used for returning the value from any function that returns an integer or pointer.

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

1. The caller, P, can save the value y.
2. P can store the value in a callee-save register (saved and restored).
Example procedure call

```c
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;
} // callswap.c and figure 3.23
```

```assembly
caller:
pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $534, -4(%ebp)
movl $1057, -8(%ebp)
leal -8(%ebp), %eax
movl %eax, 4(%esp)
leal -4(%ebp), %eax
movl %eax, (%esp)
call swap_add
movl -4(%ebp), %edx
subl -8(%ebp), %edx
imull %edx, %eax
leave
ret

swap_add:
pushl %ebp
movl %esp, %ebp
pushl %ebx
movl 8(%ebp), %ebx
movl 12(%ebp), %ecx
movl (%ebx), %eax
movl (%ecx), %edx
movl %edx, (%ebx)
movl %eax, (%ecx)
leal (%edx,%ecx), %eax
popl %ebx
popl %ebp
ret```


Stack frames for caller and swap_add

Fig 3.24