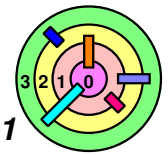
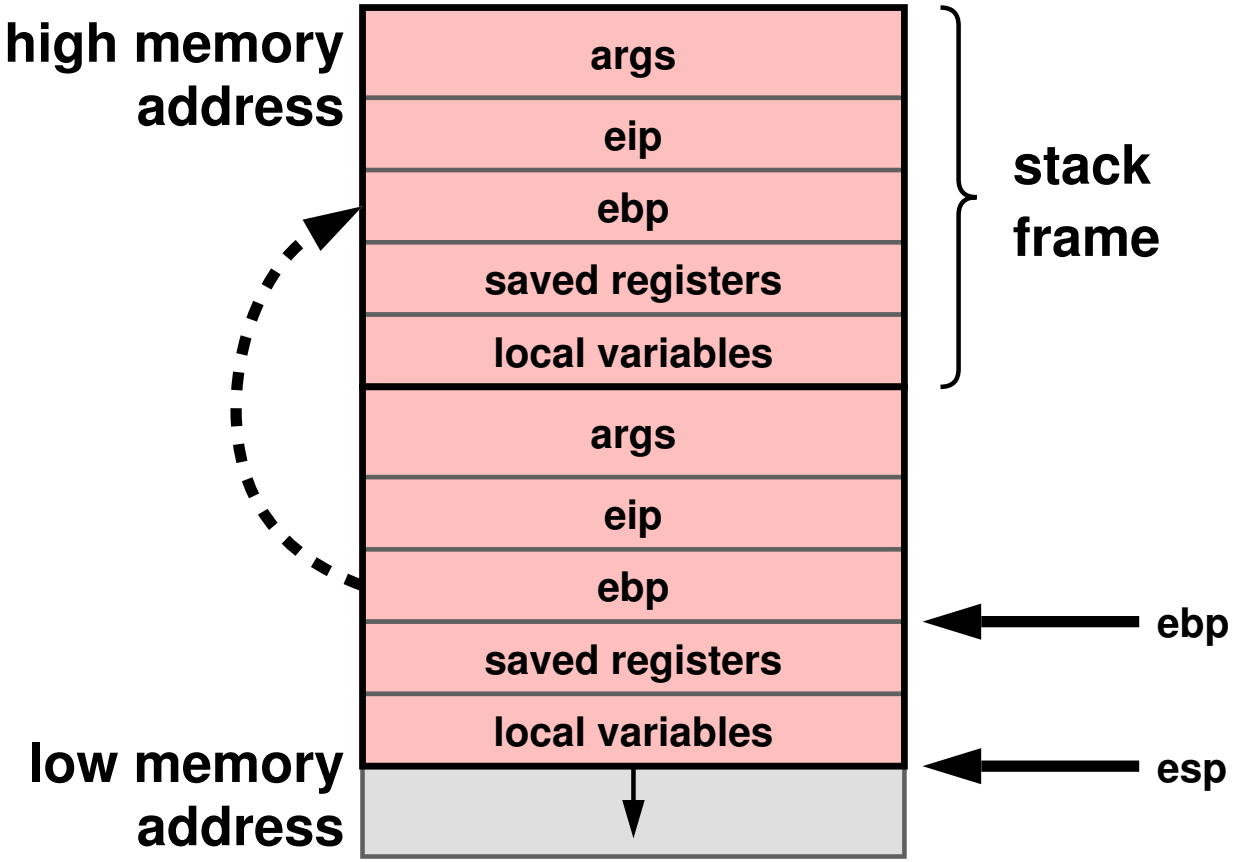
Housekeeping (Lecture 8 - 9/23/2013)

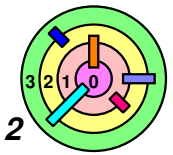
- Warmup #2 due at 11:45pm on Friday, 10/4/2013
  - if you have code from a previous semester, be very careful and *not copy any code from it*
    - it's best if you just get rid of it
  - get started soon
    - if you are stuck, make sure you come to see the TAs, the course producer, or me during office hours
- *Grading guidelines* is the *ONLY* way we will grade and we will grade on `nunki.usc.edu` in our grading account (which you don't have access to)
  - it's a good idea to run your code against the grading guidelines
- Have you installed *Ubuntu 11.10* on your laptop/desktop?
- Do you have partners for kernel assignments?
  - work with your potential partners for warmup 2
    - again, work at high level and must *not* share code



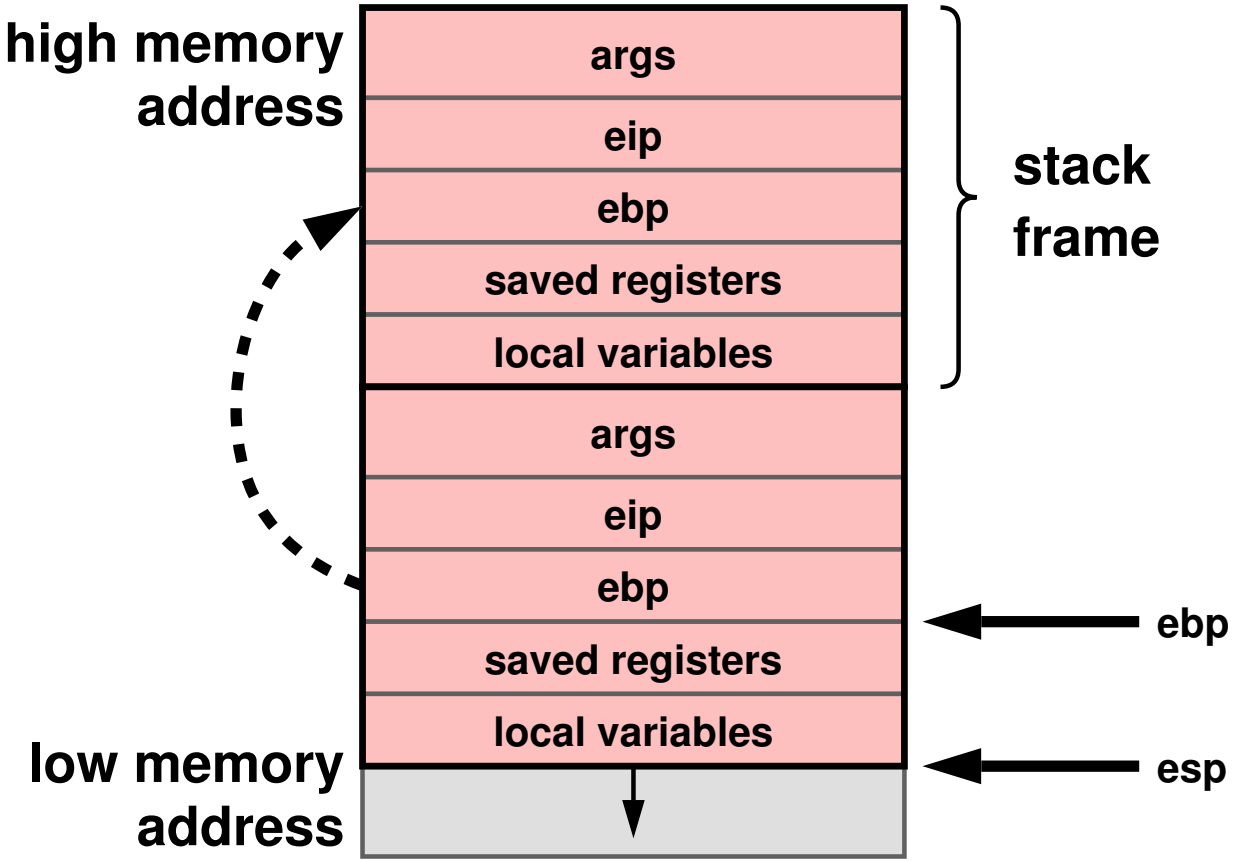
Intel x86 (32-Bit): Subroutine Linkage



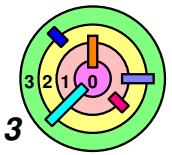
- *eip* contains address of caller's *instruction pointer* register
  - this is the *return address!*
- *ebp* contains the caller's *base (frame) pointer* register
  - this is a link to the caller's *stack frame*



Intel x86 (32-Bit): Subroutine Linkage

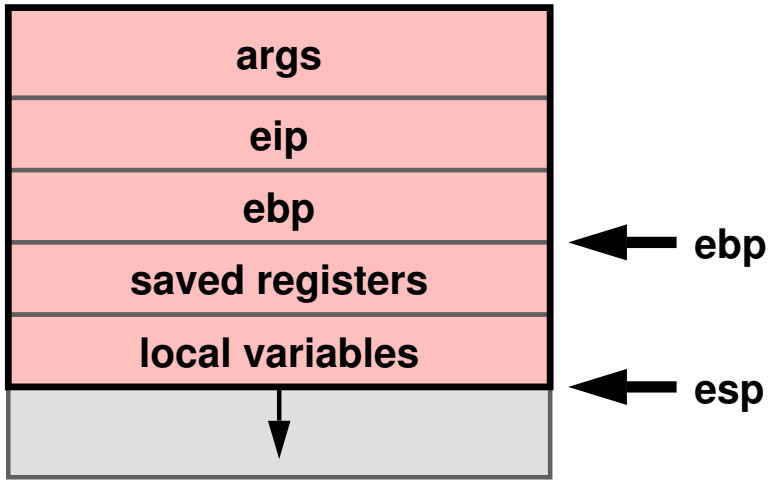


- *esp* points to the end of the current stack frame
  - it is used to prepare the next stack frame
- *eax* contains the return value of a function
- some fields are not alwasy present, compiler decides



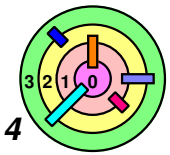
Intel x86 (32-Bit): Subroutine Linkage

- Who sets what?
  - ***args*** is explicitly setup by the ***caller***
  - ***eip*** is copied into the stack frame by a "`call`" machine instruction
  - ***ebp*** is copied explicitly by the ***callee***
  - registers are saved explicitly by the ***callee*** code
    - as it turned out, for x86, some registers are designated to be saved by the callee code
  - space for local variables is ***created*** explicitly by the ***callee*** code
    - as well as initialization of these variables



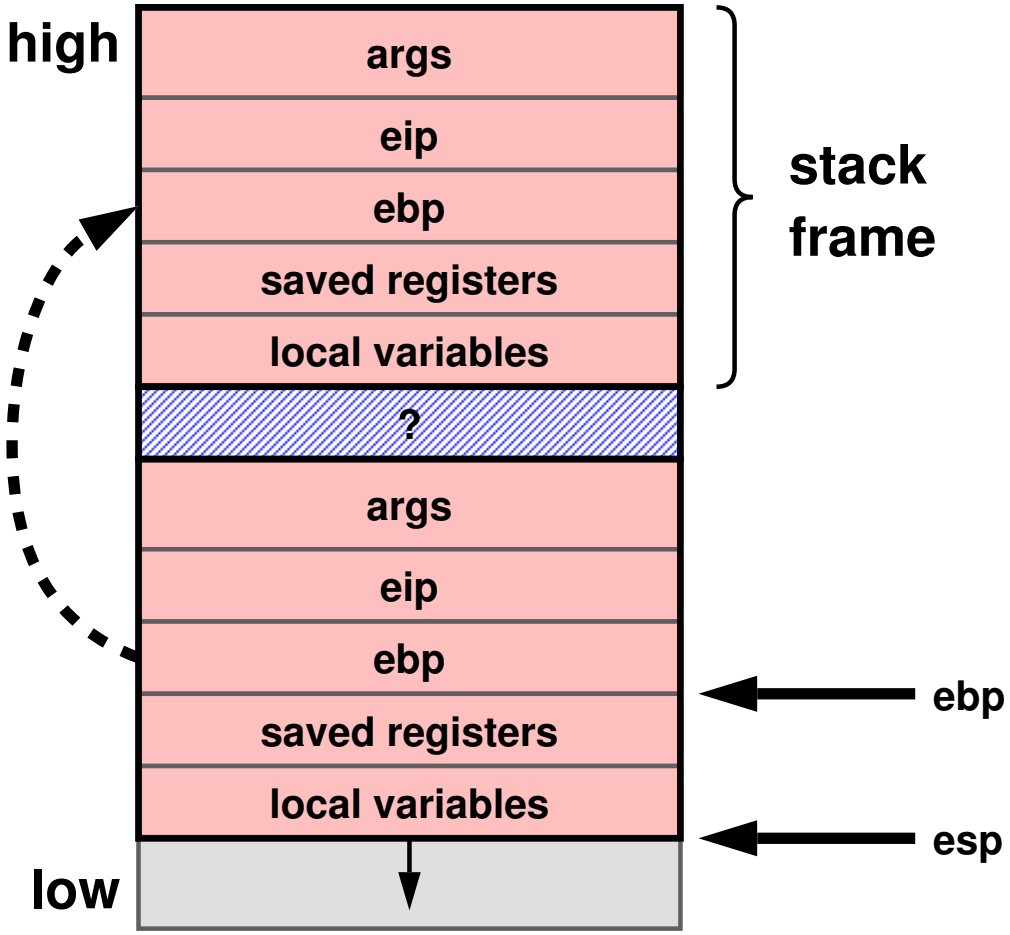
- What does the stack frame look like for the following function?

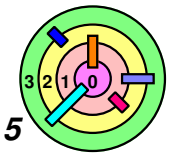
```
void func() { printf("I'm here.\n"); }
```



Intel x86 (32-Bit): Subroutine Linkage

- In reality, there can be stuff between stack frames
  - e.g., by convention, specific registers are saved and restored by the caller (this can depend on the compiler)





Intel x86: Subroutine Code (1)

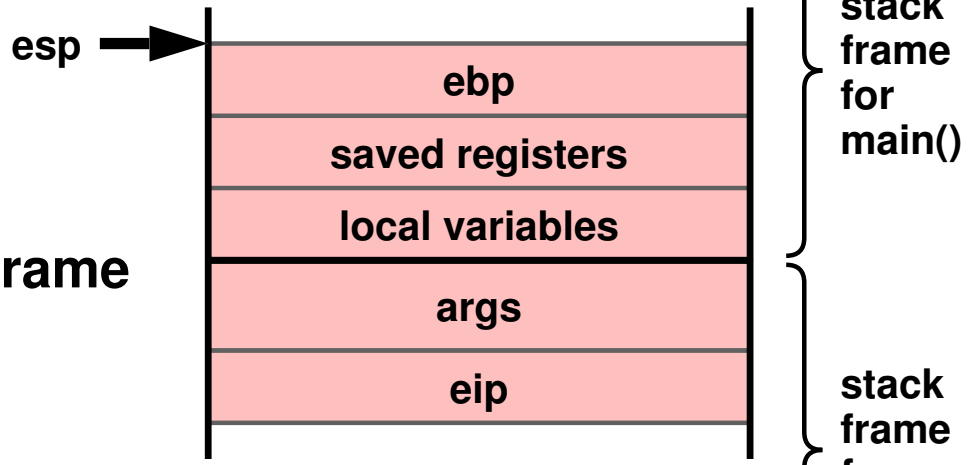
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

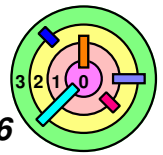
push args

pop args;
get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

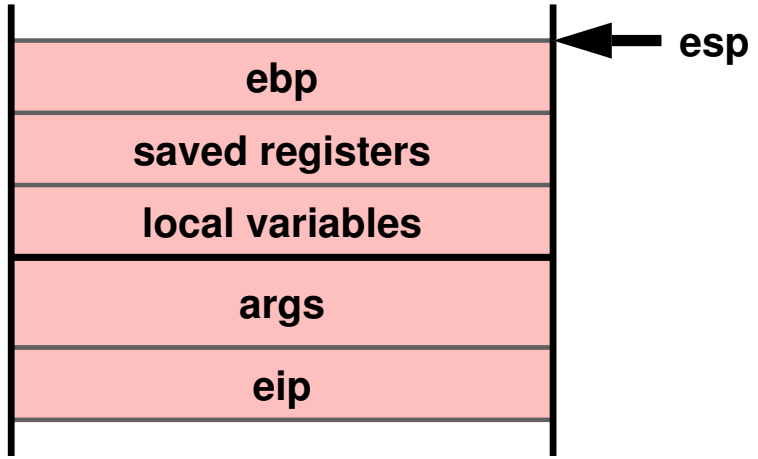
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

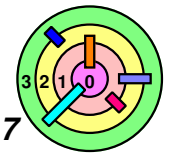
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

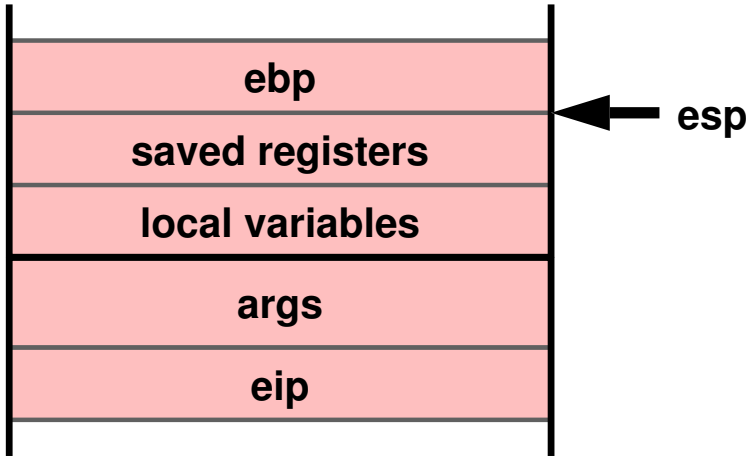
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

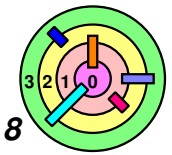
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

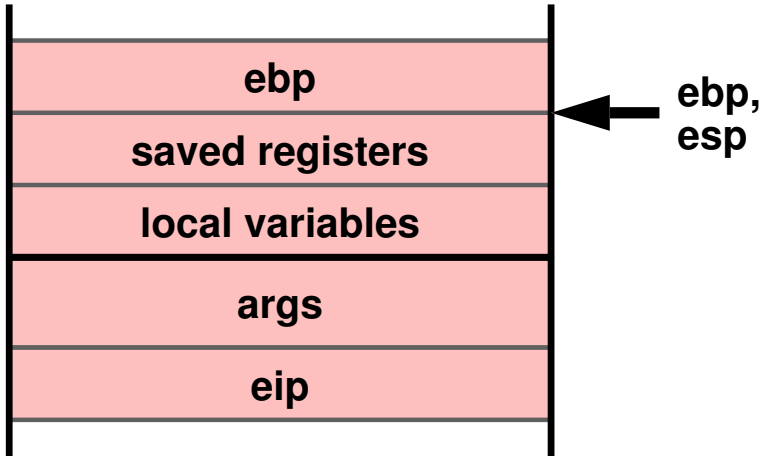
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

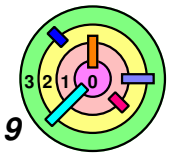
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

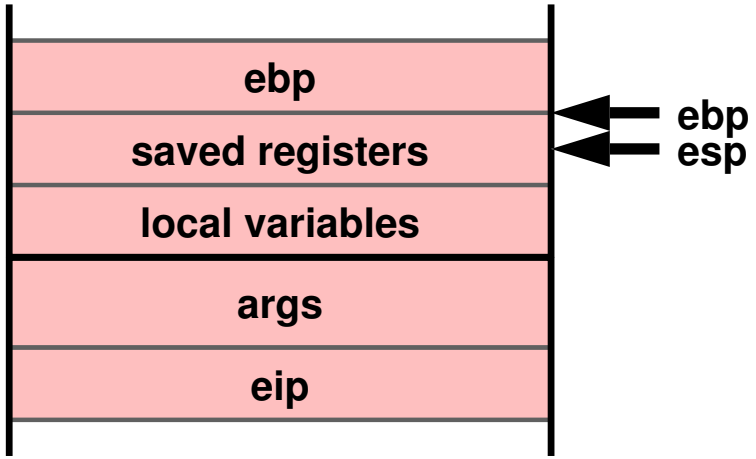
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

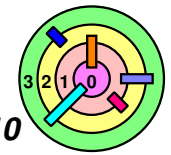
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

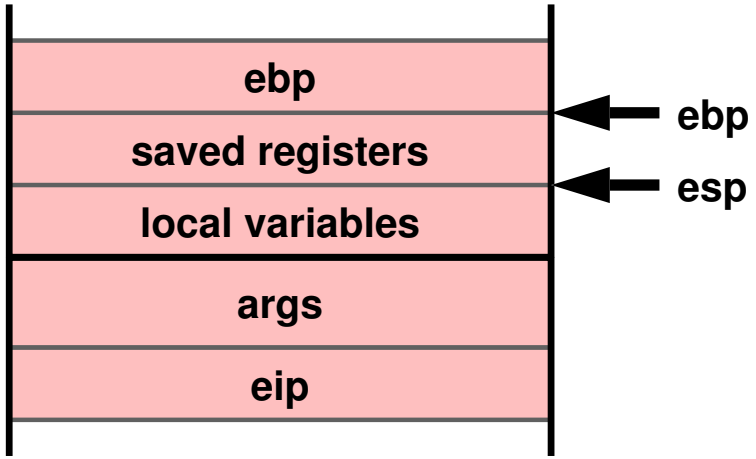
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

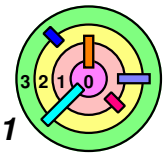
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

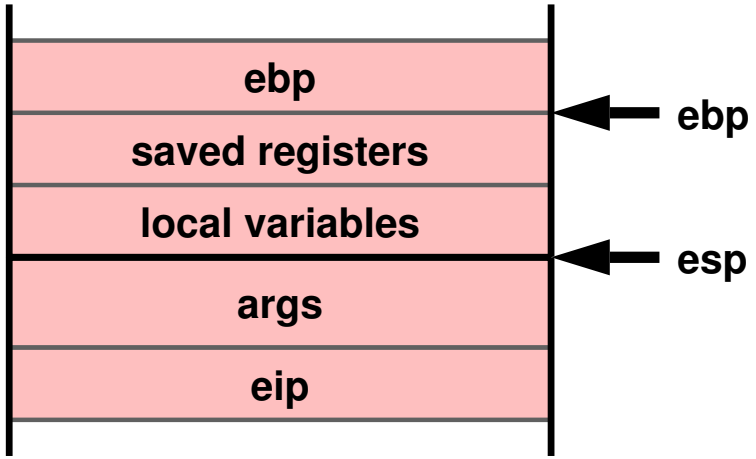
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

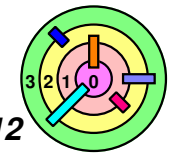
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

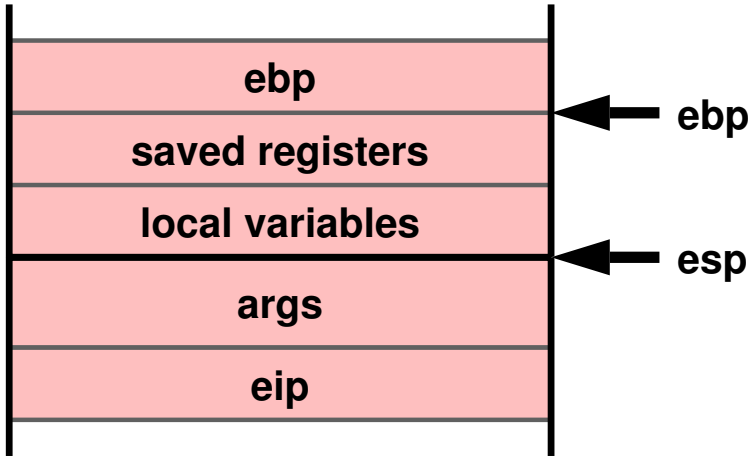
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

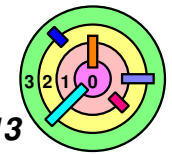
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

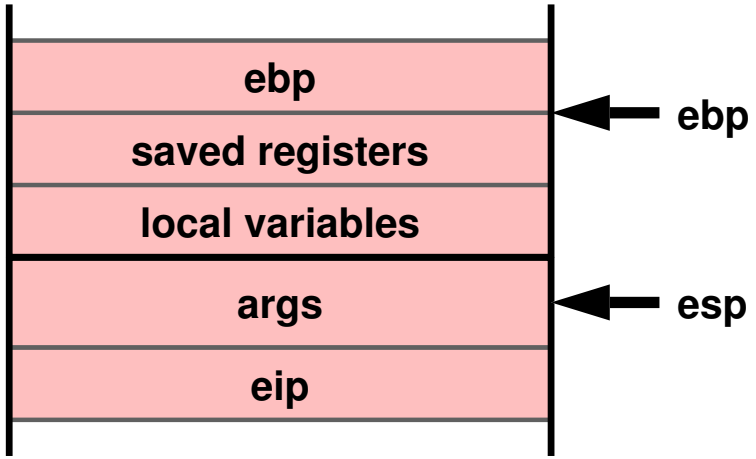
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

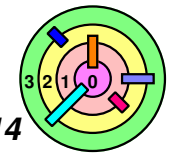
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

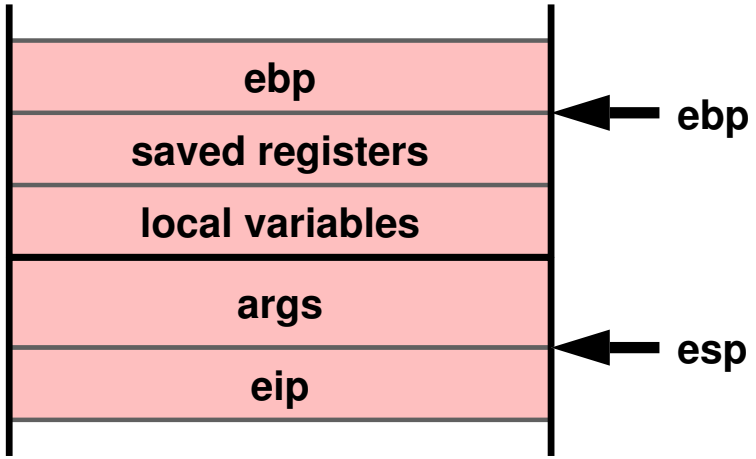
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

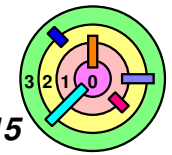
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

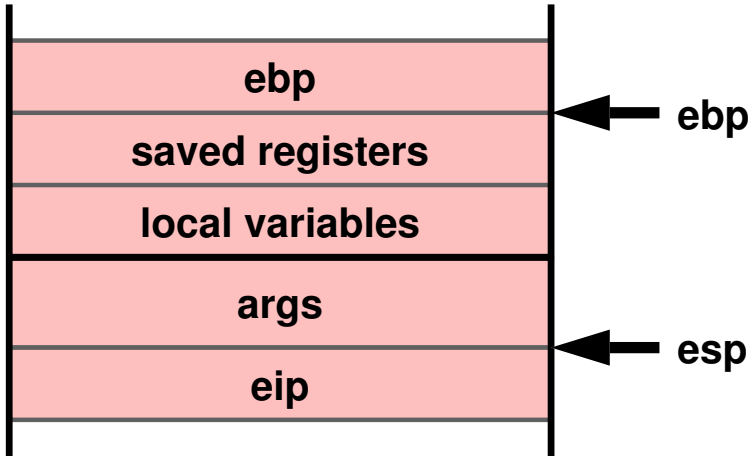
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

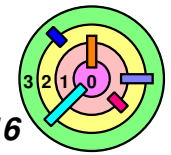
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

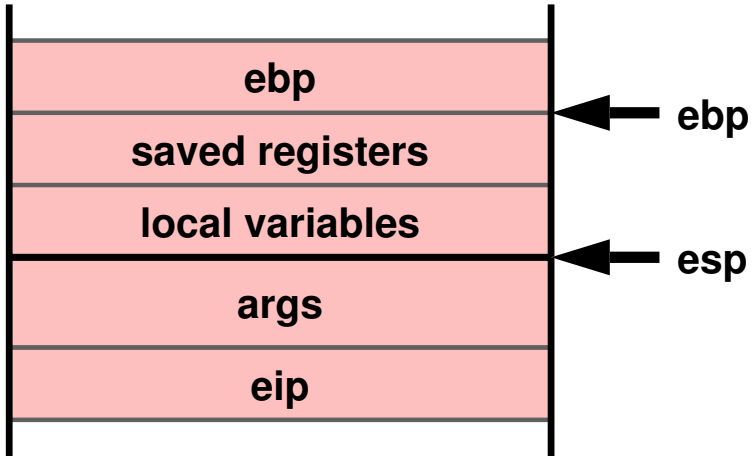
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

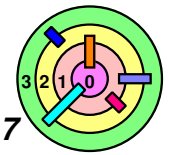
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

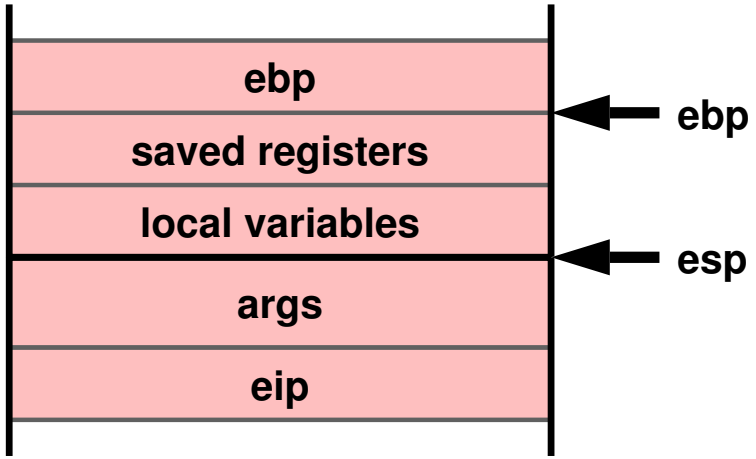
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

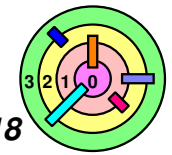
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

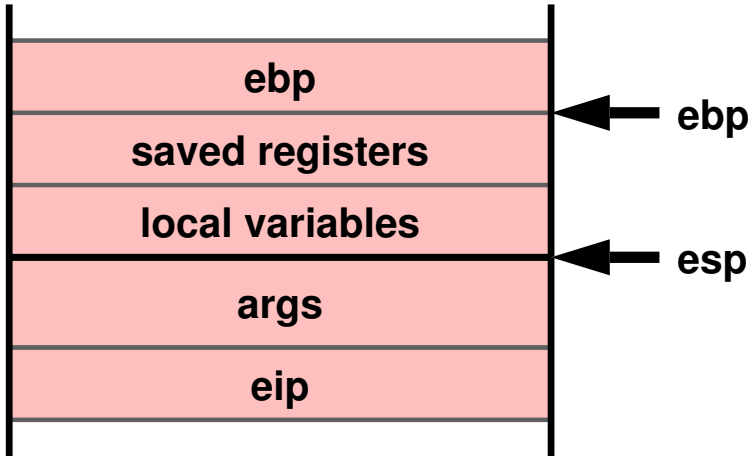
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

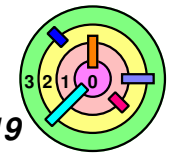
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

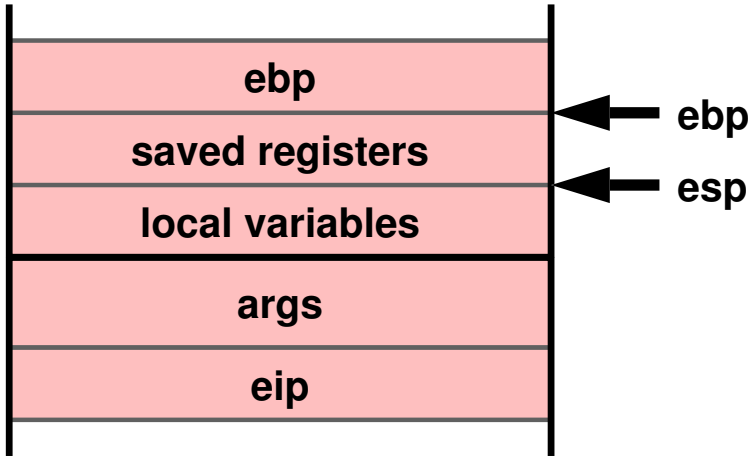
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

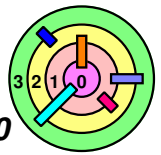
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

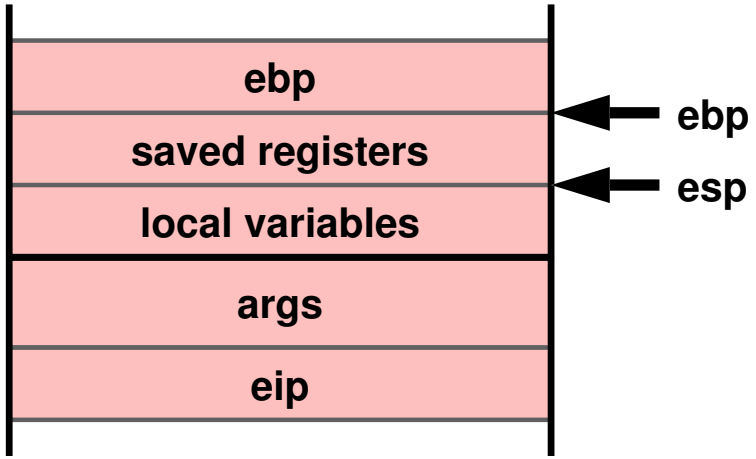
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

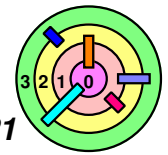
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

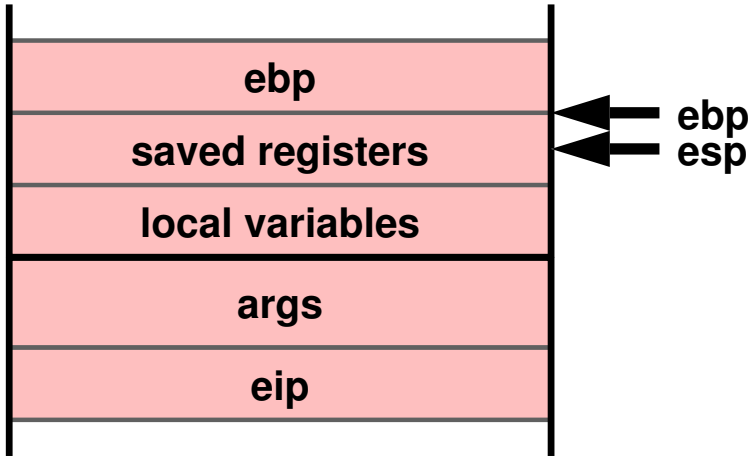
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

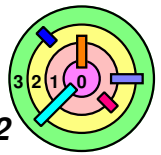
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

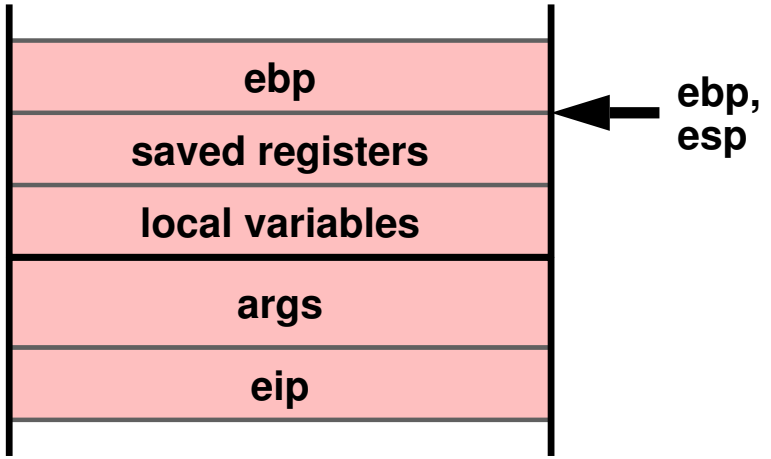
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

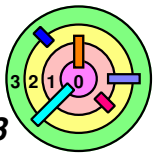
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

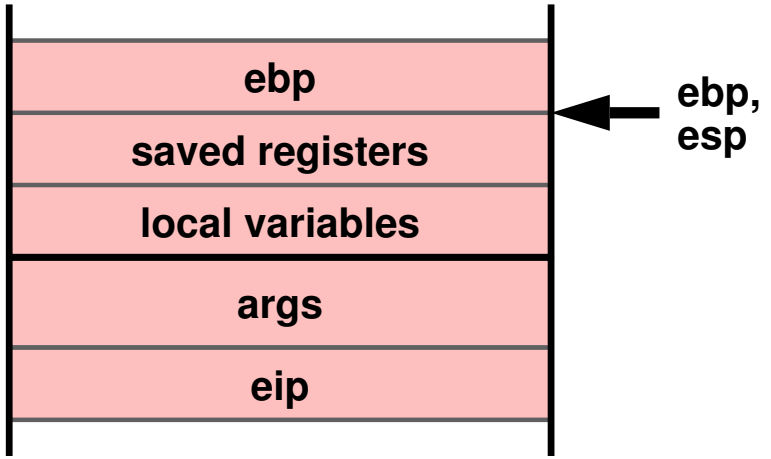
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

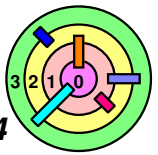
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



Intel x86: Subroutine Code (1)

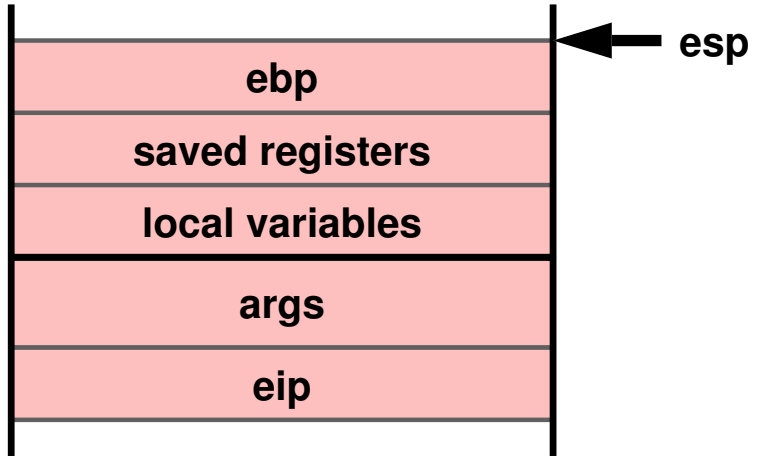
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

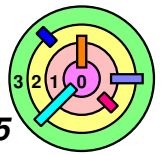
push args

pop args; get result

set return value and restore frame




```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```

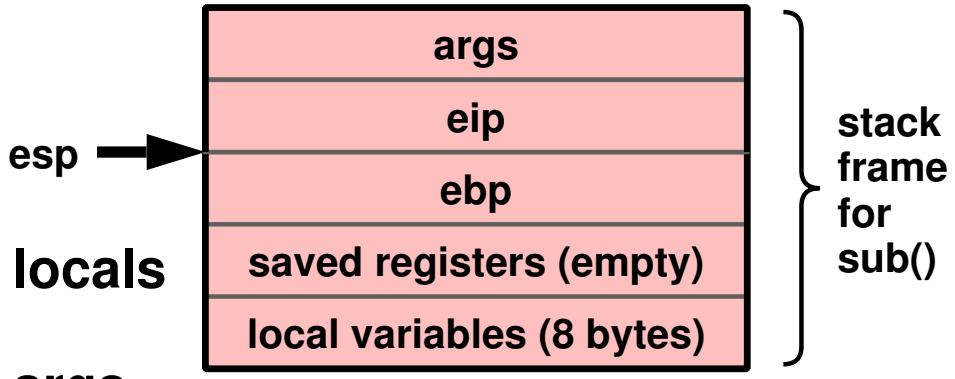


Intel x86: Subroutine Code (2)

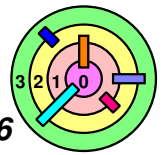
```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)
   movl $0, -8(%ebp)
   movl -4(%ebp), %ecx
   movl -8(%ebp), %eax
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```

init locals

get args




```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

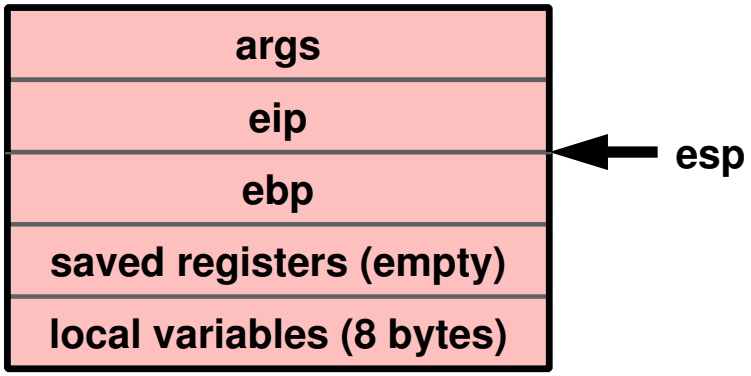


Intel x86: Subroutine Code (2)

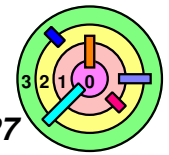
```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)
   movl $0, -8(%ebp)
   movl -4(%ebp), %ecx
   movl -8(%ebp), %eax
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```

init locals

get args




```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

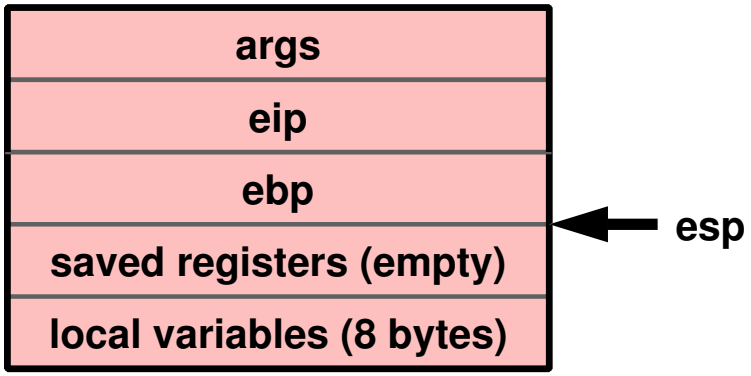


Intel x86: Subroutine Code (2)

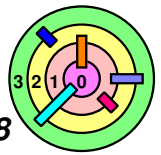
```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)
   movl $0, -8(%ebp)
   movl -4(%ebp), %ecx
   movl -8(%ebp), %eax
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```

init locals

get args




```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

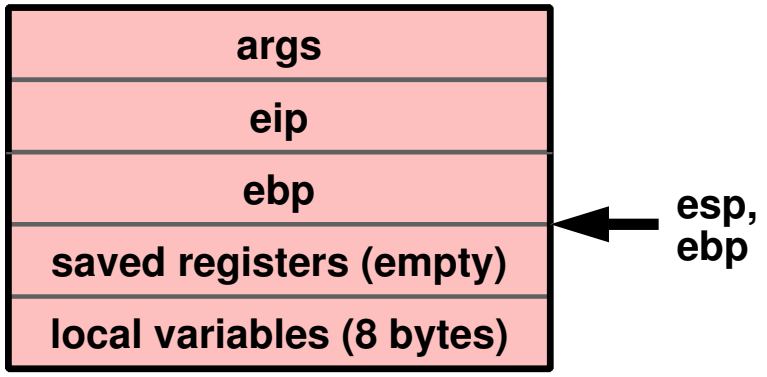


Intel x86: Subroutine Code (2)

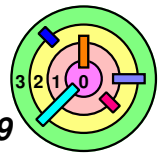
```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)
   movl $0, -8(%ebp)
   movl -4(%ebp), %ecx
   movl -8(%ebp), %eax
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```

init locals

get args




```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

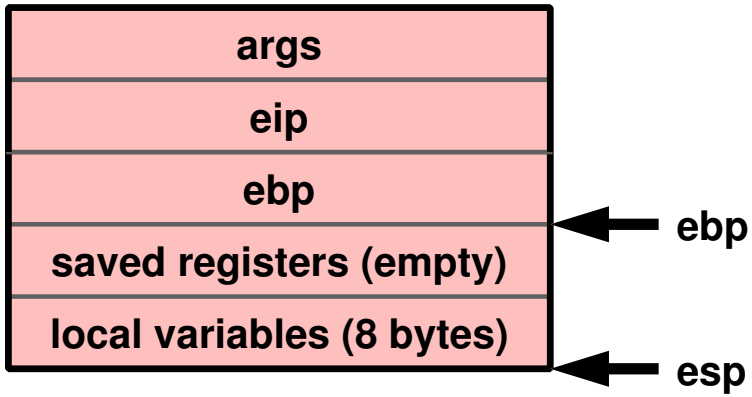


Intel x86: Subroutine Code (2)

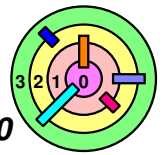
```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)
   movl $0, -8(%ebp)
   movl -4(%ebp), %ecx
   movl -8(%ebp), %eax
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```

init locals

get args




```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

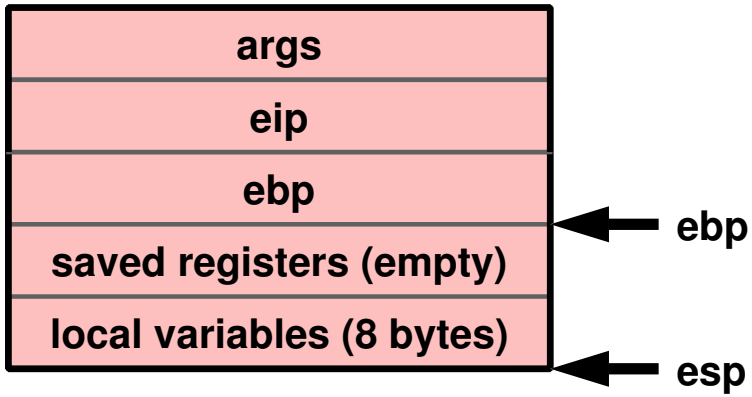


Intel x86: Subroutine Code (2)

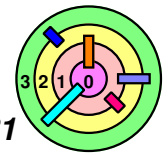
```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)
   movl $0, -8(%ebp)
   movl -4(%ebp), %ecx
   movl -8(%ebp), %eax
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```

init locals

get args




```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

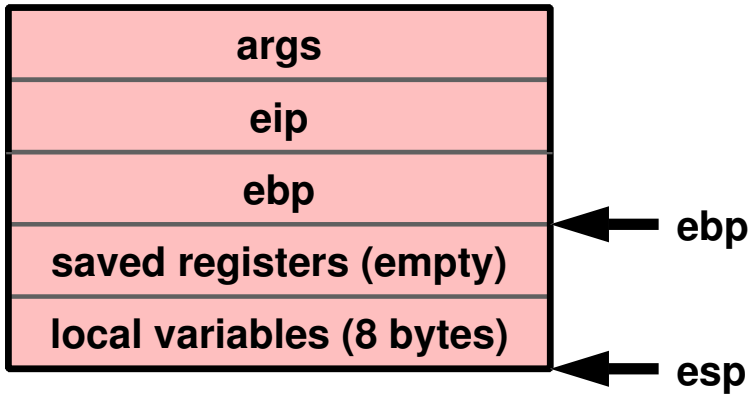


Intel x86: Subroutine Code (2)

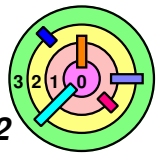
```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)
   movl $0, -8(%ebp)
   movl -4(%ebp), %ecx
   movl -8(%ebp), %eax
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```

init locals

get args

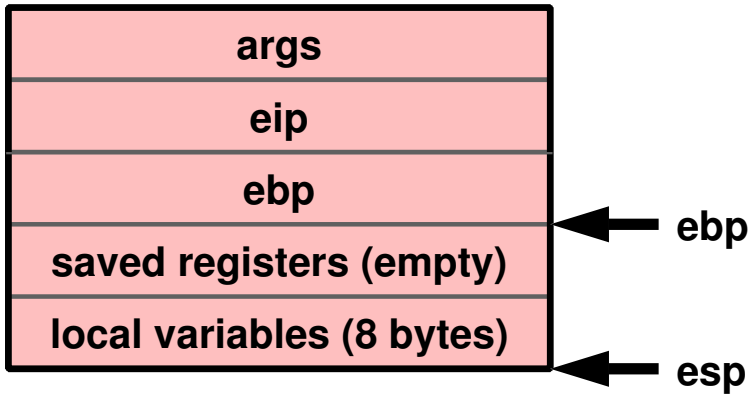



```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

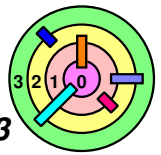


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)        } init locals
   movl $0, -8(%ebp)        }
   movl -4(%ebp), %ecx      } get args
   movl -8(%ebp), %eax      }
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```




```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

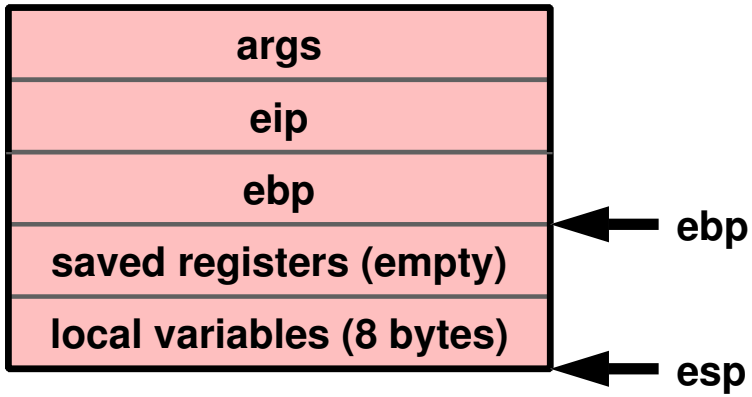


Intel x86: Subroutine Code (2)

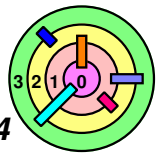
```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)
   movl $0, -8(%ebp)
   movl -4(%ebp), %ecx
   movl -8(%ebp), %eax
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```

init locals

get args

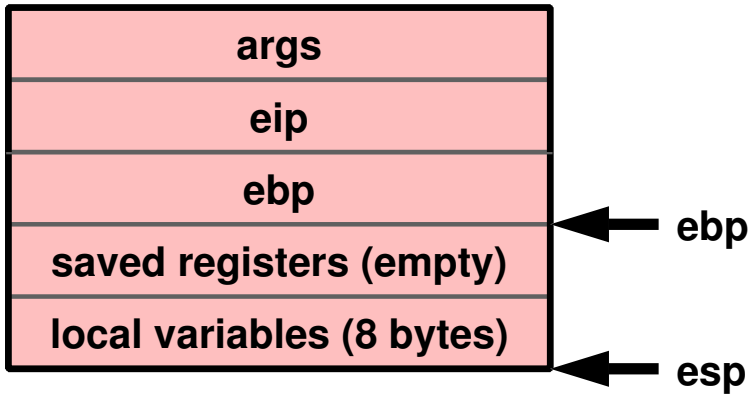



```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

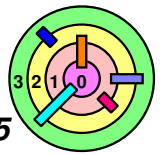


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)        } init locals
   movl $0, -8(%ebp)        }
   movl -4(%ebp), %ecx      } get args
   movl -8(%ebp), %eax      }
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```




```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

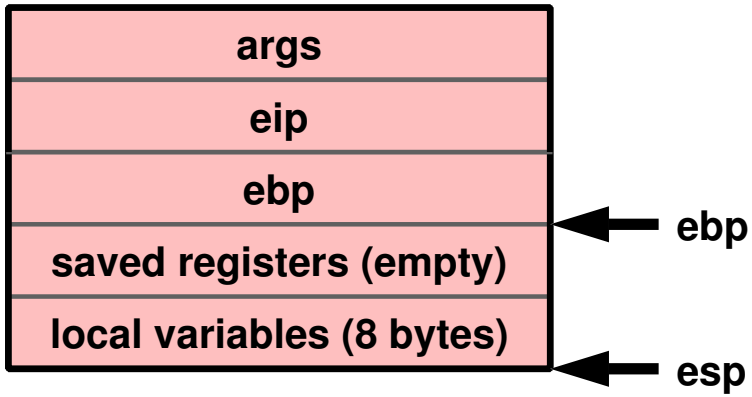


Intel x86: Subroutine Code (2)

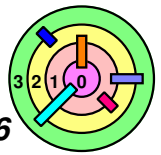
```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)
   movl $0, -8(%ebp)
   movl -4(%ebp), %ecx
   movl -8(%ebp), %eax
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```

init locals

get args

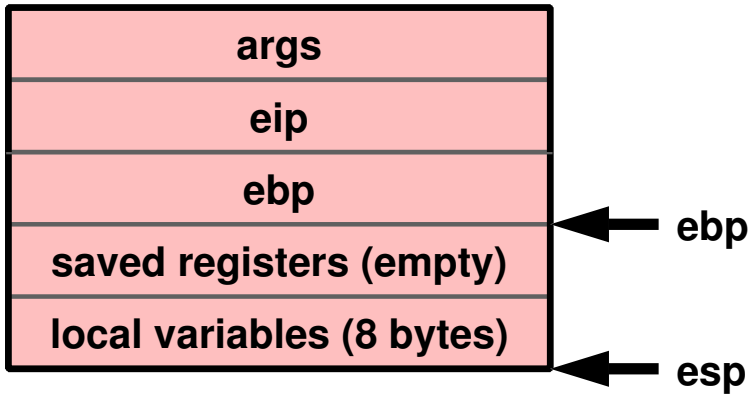



```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

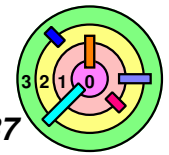


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)        } init locals
   movl $0, -8(%ebp)        }
   movl -4(%ebp), %ecx      } get args
   movl -8(%ebp), %eax      }
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```

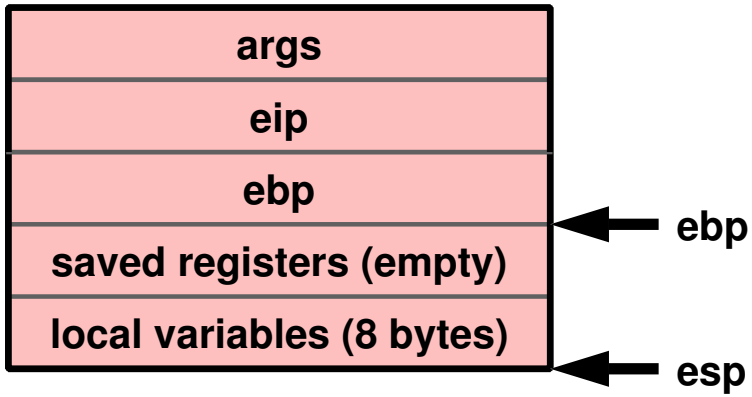



```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

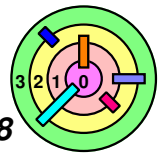


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)        } init locals
   movl $0, -8(%ebp)        }
   movl -4(%ebp), %ecx      } get args
   movl -8(%ebp), %eax      }
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```




```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

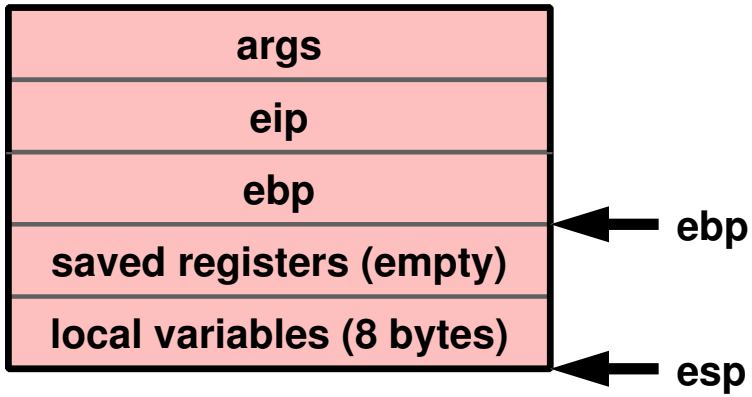


Intel x86: Subroutine Code (2)

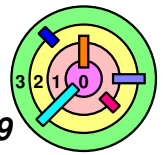
```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)
   movl $0, -8(%ebp)
   movl -4(%ebp), %ecx
   movl -8(%ebp), %eax
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```

init locals

get args




```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

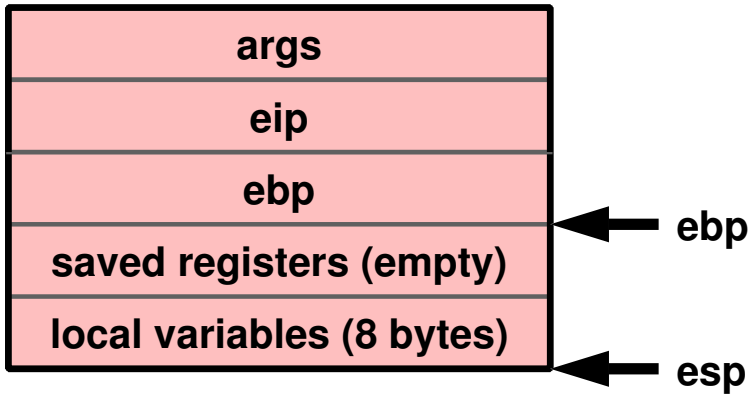


Intel x86: Subroutine Code (2)

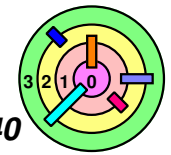
```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)
   movl $0, -8(%ebp)
   movl -4(%ebp), %ecx
   movl -8(%ebp), %eax
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```

init locals

get args

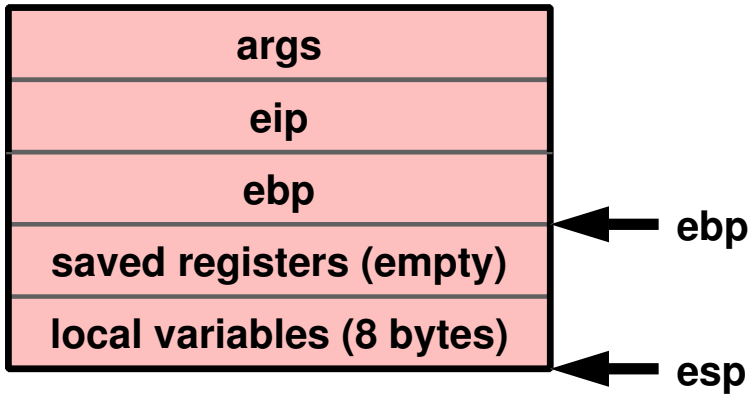



```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

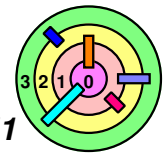


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)        } init locals
   movl $0, -8(%ebp)        }
   movl -4(%ebp), %ecx      } get args
   movl -8(%ebp), %eax      }
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```




```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

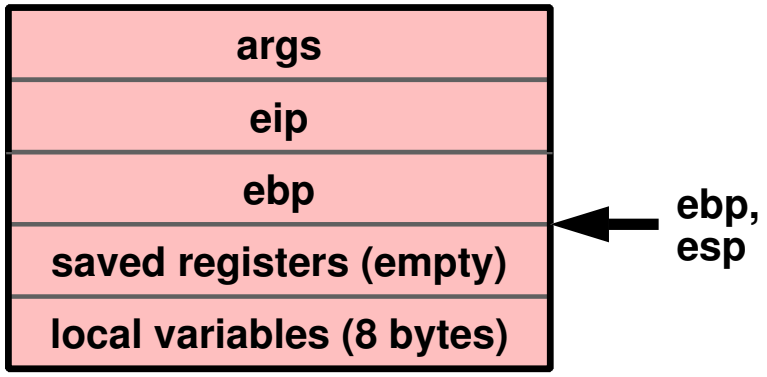


Intel x86: Subroutine Code (2)

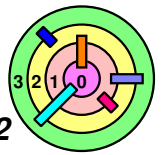
```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)
   movl $0, -8(%ebp)
   movl -4(%ebp), %ecx
   movl -8(%ebp), %eax
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```

init locals

get args




```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

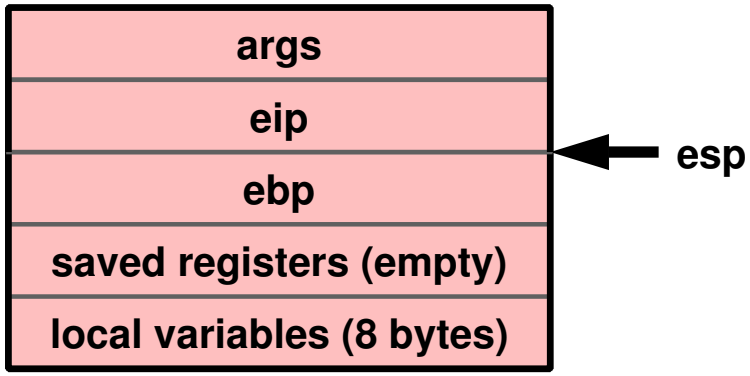


Intel x86: Subroutine Code (2)

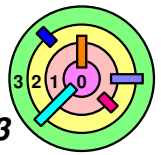
```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)
   movl $0, -8(%ebp)
   movl -4(%ebp), %ecx
   movl -8(%ebp), %eax
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```

init locals

get args




```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```



SPARC Architecture

return address	i7	r31
frame pointer	i6	r30
	i5	r29
	i4	r28
	i3	r27
	i2	r26
	i1	r25
	i0	r24

Input Registers

	o7	r15
stack pointer	o6	r14
	o5	r13
	o4	r12
	o3	r11
	o2	r10
	o1	r9
	o0	r8

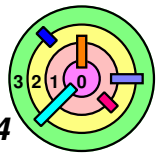
Output Registers

	l7	r23
	l6	r22
	l5	r21
	l4	r20
	l3	r19
	l2	r18
	l1	r17
	l0	r16

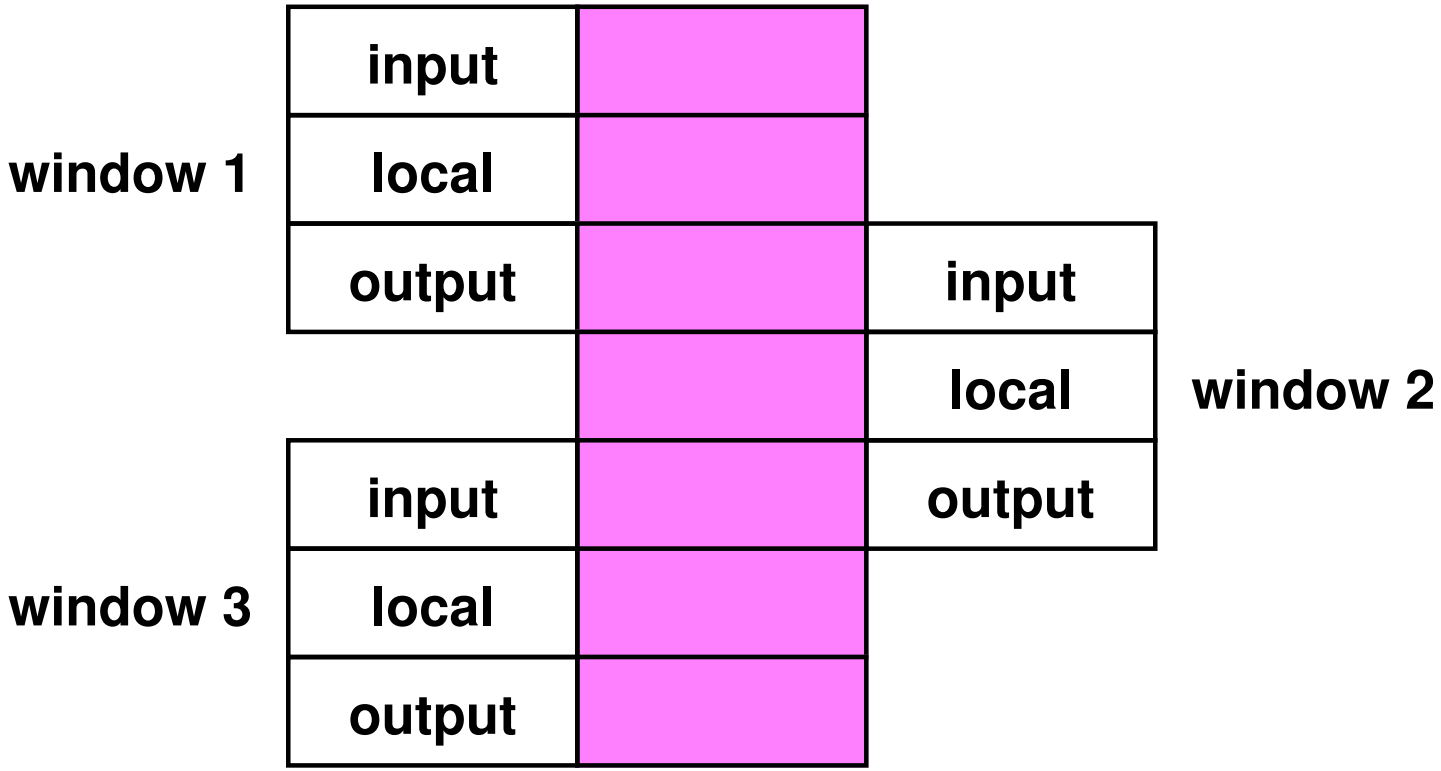
Local Registers

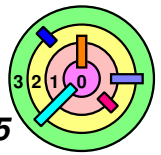
	g7	r7
	g6	r6
	g5	r5
	g4	r4
	g3	r3
	g2	r2
	g1	r1
0	g0	r0

Global Registers

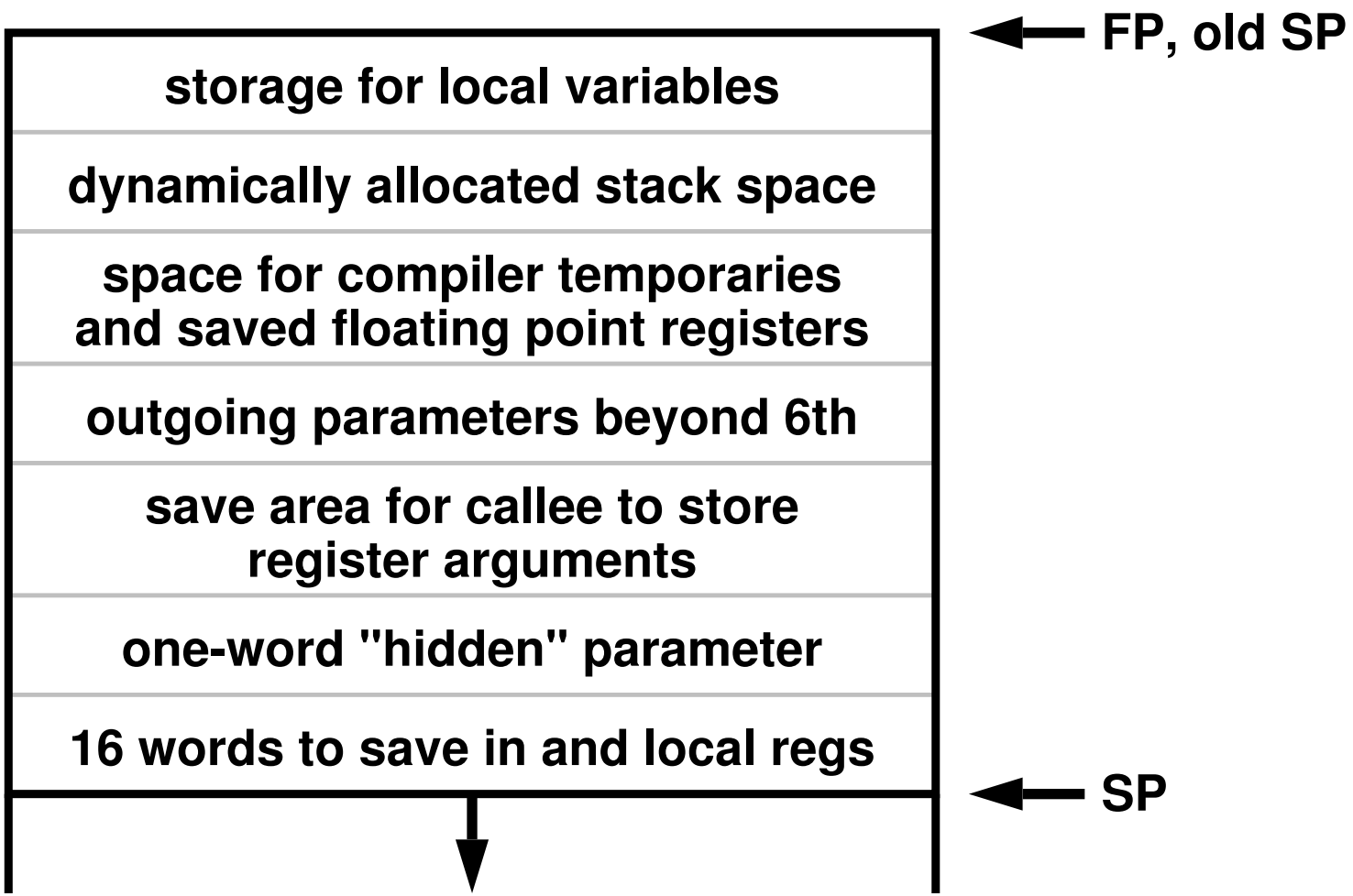


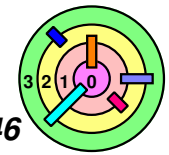
SPARC Architecture: Register Windows





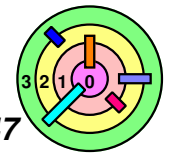
SPARC Architecture: Stack





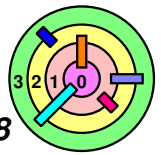
SPARC Architecture: Subroutine Code

```
ld [%fp-8], %o0
! put local var (a) into out register
mov 1, %o1
! deal with 2nd parameter
call sub
nop
st  %o0, [%fp-4]
! store result into local var (i)
...
sub:
save %sp, -64, %sp
! push a new stack frame
add  %i0, %i1, %i0
! compute sum
ret
! return to caller
restore
! pop frame off stack (in delay slot)
```

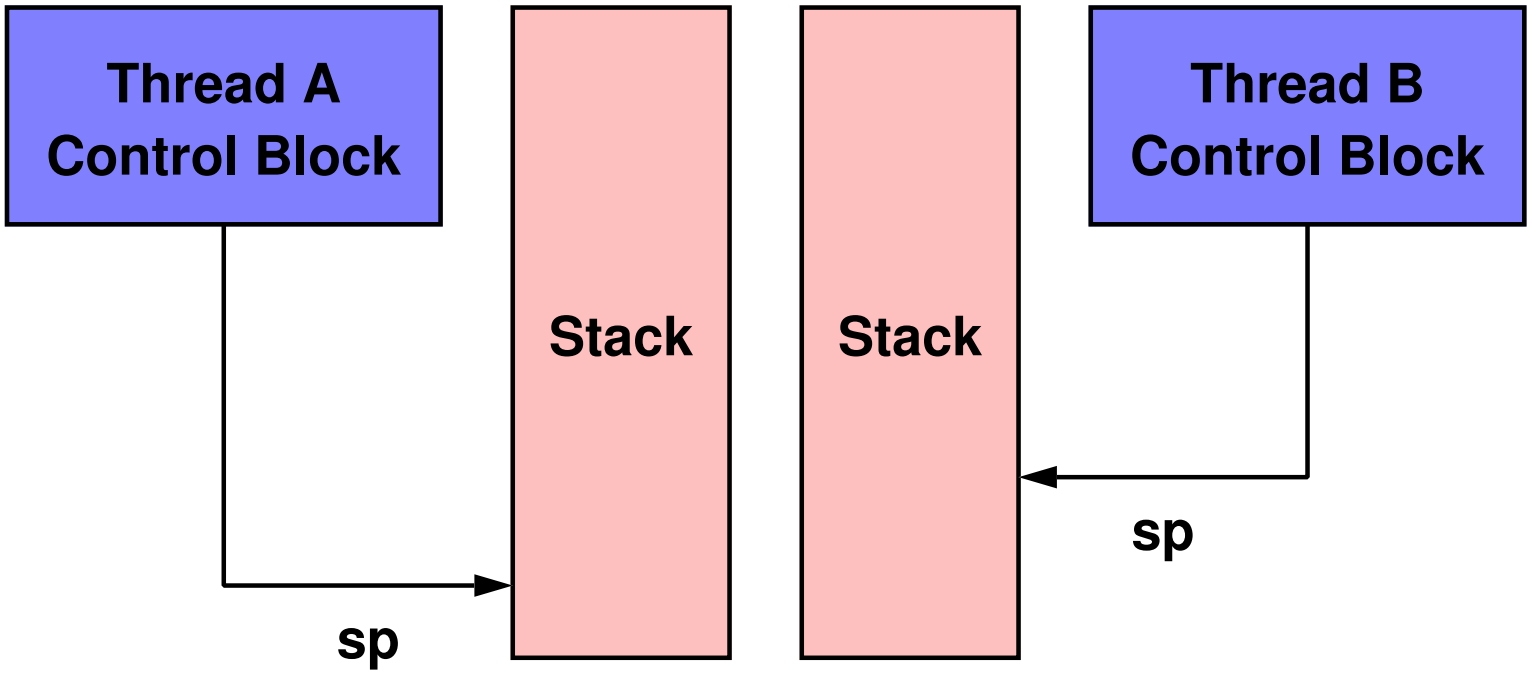


3.1 Context Switching

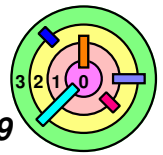
- Procedures
- *Threads & Coroutines*
- Systems Calls
- Interrupts



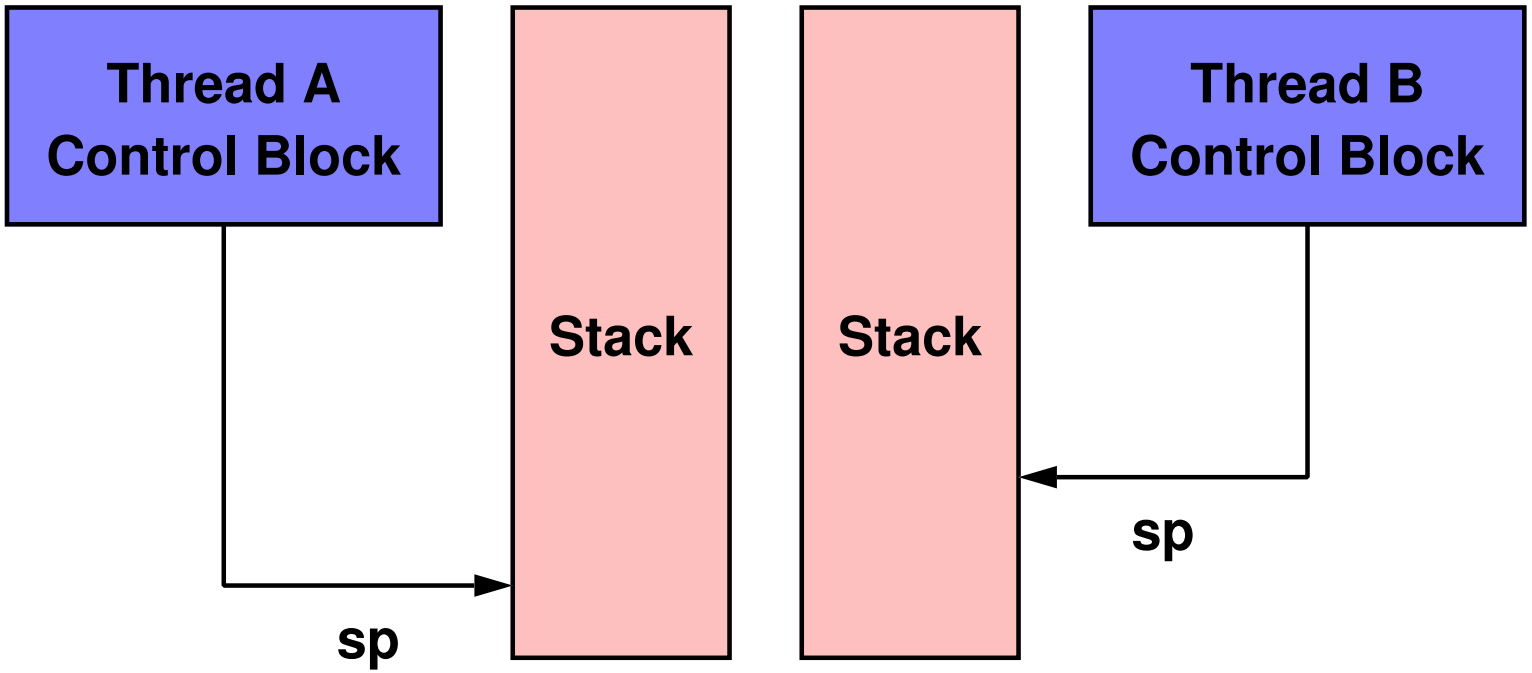
Representing Threads



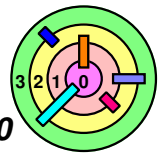
- normally, threads are independent of one another and don't directly control one another's execution
- threads can be made aware of each other and be able to *transfer control* from one thread to another
  - this is known as *coroutine linkage*



Representing Threads

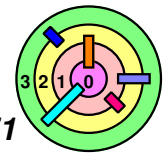


- A thread's context
  - its stack, its register state
  - can be stored in a *thread control block* (directly or indirectly)
- To transfer control from one thread to another is equivalent to copying the thread control block of the target thread into the current thread context



Switching Between Threads

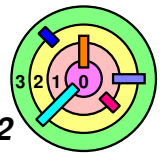
```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```



Switching Between Threads

```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```

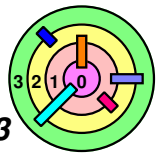
```
switch:
  ;enter switch, creating new stack frame
  pushl %ebp ;push FP
  movl  %esp,%ebp ;set FP to point to new frame
  pushl %esi  ;save esi register
  movl  CurrentThread,%esi ;load address of caller's TCB
  movl  %esp,SP(%esi) ;save SP in control block
  movl  8(%ebp),CurrentThread ;store target TCB address
                              ;into CurrentThread
  movl  CurrentThread,%esi ;put new TCB address into esi
  movl  SP(%esi),%esp ;restore target thread's SP
  ;we're now in the context of the target thread!
  popl  %esi ;restore target thread's esi register
  popl  %ebp ;pop target threadss FP
  ret ;return to caller within target thread
```



Switching Between Threads

```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```

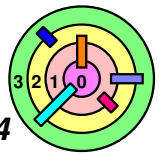
- on entry into `switch()`, the caller's registers are saved!



Switching Between Threads

```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```

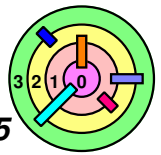
- on entry into `switch()`, the caller's registers are saved!
- then the current stack pointer is saved into current thread's thread control block



Switching Between Threads

```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```

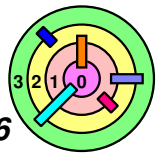
- on entry into `switch()`, the caller's registers are saved!
- then the current stack pointer is saved into current thread's thread control block
- the thread control block of the target thread is copied into the current thread context



Switching Between Threads

```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```

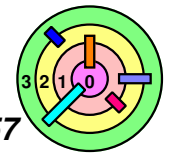
- on entry into `switch()`, the caller's registers are saved!
- then the current stack pointer is saved into current thread's thread control block
- the thread control block of the target thread is copied into the current thread context
- fetch the target thread's stack pointer from its thread control block and loads it into the actual stack pointer



Switching Between Threads

```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```

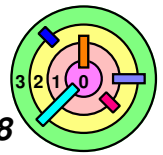
- on entry into `switch()`, the caller's registers are saved!
- then the current stack pointer is saved into current thread's thread control block
- the thread control block of the target thread is copied into the current thread context
- fetch the target thread's stack pointer (esp for x86) from its thread control block and loads it into the actual stack pointer
- on return from `switch()`, the registers (ebp and eip for x86) are restored into the current thread, which is the target thread!



Switching Between Threads

```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```

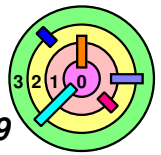
- on entry into `switch()`, the caller's registers are saved!
- then the current stack pointer is saved into current thread's thread control block
- the thread control block of the target thread is copied into the current thread context
- fetch the target thread's stack pointer (esp for x86) from its thread control block and loads it into the actual stack pointer
- on return from `switch()`, the registers (ebp and eip for x86) are restored into the current thread, which is the target thread!
- if thread control blocks were user-space data structures, threads were switched *without* getting the kernel involved!



Switching Between Threads

```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```

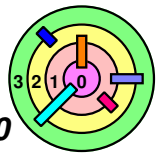
- Note: one very interesting thing happened in this call
  - usually, a single thread executes the entire procedure call
  - with `switch()`, at the beginning of the procedure call, one thread is executing
    - half way through the procedure call, another thread starts to execute
    - so, one thread enters the `switch()` call, and a different thread leaves the `switch()` call!
- This is an elegant way of switching threads
  - all threads come here to switch to another thread



... in x86 Assembler

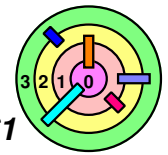
```
switch:
  ;enter switch, creating ne
  pushl %ebp ;push FP
  movl  %esp,%ebp ;set FP to
  pushl %esi  ;save esi regi
  movl  CurrentThread,%esi ;
  movl  %esp,SP(%esi) ;save
  movl  8(%ebp),CurrentThread ;store target TCB address
                              ;into CurrentThread
  movl  CurrentThread,%esi ;put new TCB address into esi
  movl  SP(%esi),%esp ;restore target thread's SP
  ;we're now in the context of the target thread!
  popl  %esi ;restore target thread's esi register
  popl  %ebp ;pop target threadss FP
  ret ;return to caller within target thread
```

```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```



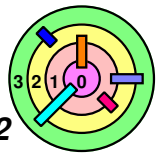
... in SPARC Assembler

```
switch:
  save %sp, -64, %sp   ! Push a new stack frame.
  t    3               ! Trap into the OS to force
                       !     window overflow.
  st   %sp, [%g0+SP]   ! Save CurrentThread's SP in
                       !     control block.
  mov  %i0, %g0        ! Set CurrentThread to be
                       !     target thread.
  ld   [%g0+SP], %sp   ! Set SP to that of target thread
  ret                  ! return to caller (in target
                       !     thread's context).
  restore              ! Pop frame off stack (in delay
                       !     slot).
```



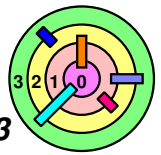
3.1 Context Switching

- Procedures
- Threads & Coroutines
- *Systems Calls*
- Interrupts



System Calls

- A system call involves the transfer of control from user code to system/kernel code and back
  - there is no thread switching!
  - a user thread *change status* and becomes a kernel thread
    - and executes in priviledged mode
    - and executing operating-system code
      - effectively, it's part of the OS
  - then it changed back to a user thread
- Most systems provide threads with two stacks
  - one for use in user mode
  - and one for use in kernel mode
    - usually, one kernel stack shared by all threads in the same user process
  - therefore, when a thread performs a system call and switches from user mode to kernel mode
    - it switches to use its kernel-mode stack

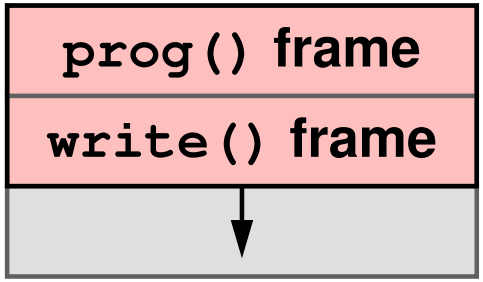


System Calls

- A *trap* is a *"software interrupt"*
  - interrupt handler will invoke trap handler

```
prog( ) {
   ...
   write(fd, buffer, size);
   ...
}
```

```
write( ) {
   ...
   trap(write_code);
   ...
}
```



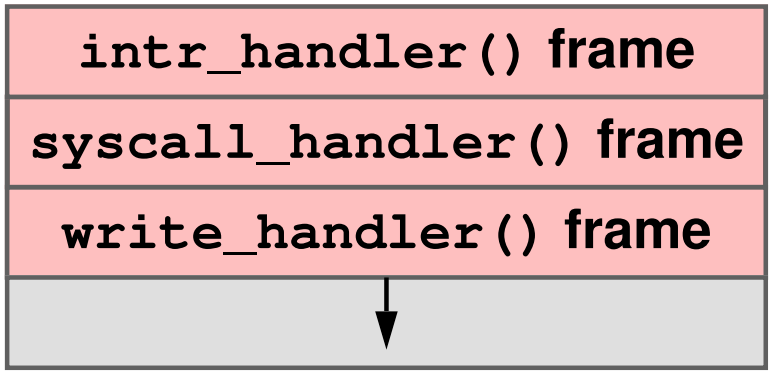
User Stack

User

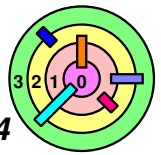
Kernel

```
intr_handler(intr_code) {
   ...
   if (intr_code == SYSCALL)
     syscall_handler( );
   ...
}

syscall_handler(trap_code) {
   ...
   if (trap_code == write_code)
     write_handler( );
   ...
}
```



Kernel Stack



System Calls

- More details on the *"trap"* machine instruction
  1) Trap into the kernel with all *interrupt disabled* and processor mode set to *kernel mode*
  2) The hardware save IP and SP in "temporary locations" in kernel space *(e.g., the interrupt stack)*
     - additional registers may be saved
  3) The hardware sets the SP to point to the *kernel stack* designated for the corresponding user process (information from PCB)
  4) HW sets IP to *interrupt* then *system call handler* (written in C)
     - pop user IP and SP from "temporary location" and push them onto kernel stack, then *re-enable interrupt*
  5) On return from the trap handler, disable interrupt and executes a special "return" instruction to *return to user process*
     - `iret` on x86
- Similar sequence happens when you get *hardware interrupt* (as we will see next)