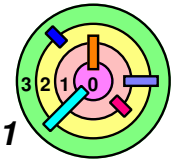
Ch 3: Basic Concepts

Bill Cheng

http://merlot.usc.edu/cs402-s16



What's Next?

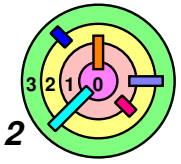
- So far, we have talked about abstractions
  - processes, files, threads
    - stuff at the user level
- We are not ready to talk about the OS yet
- Next step is something in between

Abstractions
(processes, files, threads)

- context for execution
- I/O architecture
- dynamic storage allocation
- linking & loading
- booting

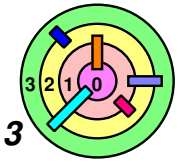
User

OS



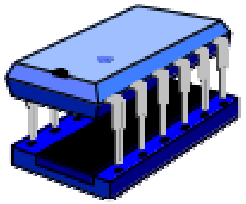
3.1 Context Switching

- Procedures
- Threads & Coroutines
- Systems Calls
- Interrupts



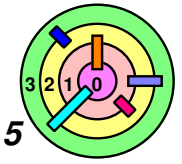
Context Switching

- The magic of OS
  - to provide the illusion that applications run concurrently and each application thinks it's the only application running on the processor
- The OS switches the processor from one application to another
  - switching happens transparently to the applications
- What is the OS doing when an application is running?



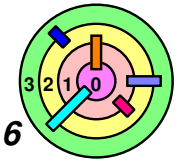
Context

- What's the execution context of a thread?
  - if we are going to talk about context switching, we need to know what we are switching and how to get back
- The *execution context* of a thread is the *current state* of our thread
  - what does the execution context include?
    - CPU registers, including the *instruction pointer*, *stack pointer*, *base/frame pointer*, etc.
    - stack
    - open files
    - etc.
    - i.e., things that may affect the execution of the thread
  - turns out the stack is complicated
    - in reality, it's just the *current stack frame* of the current thread
    - what's below it (and the rest of the address space) is also part of the thread's state



3.1 Context Switching

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

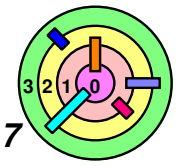


Subroutines

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

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

- You are in `main()` and are ready to call `sub()`
  - how do you make sure that `sub()` has the right context to execute the code in `sub()`?
    - you need to prepare the context for `sub()`
  - how do you make sure that you can return from `sub()` and restore the `main()` context and continue to execute properly?
    - you need to first ***save*** the context of `main()`
    - after return from `sub()`, you need to ***restore*** the context of `main()` so `main()` can ***resume*** execution

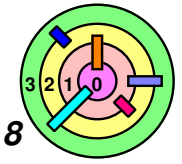


Subroutines

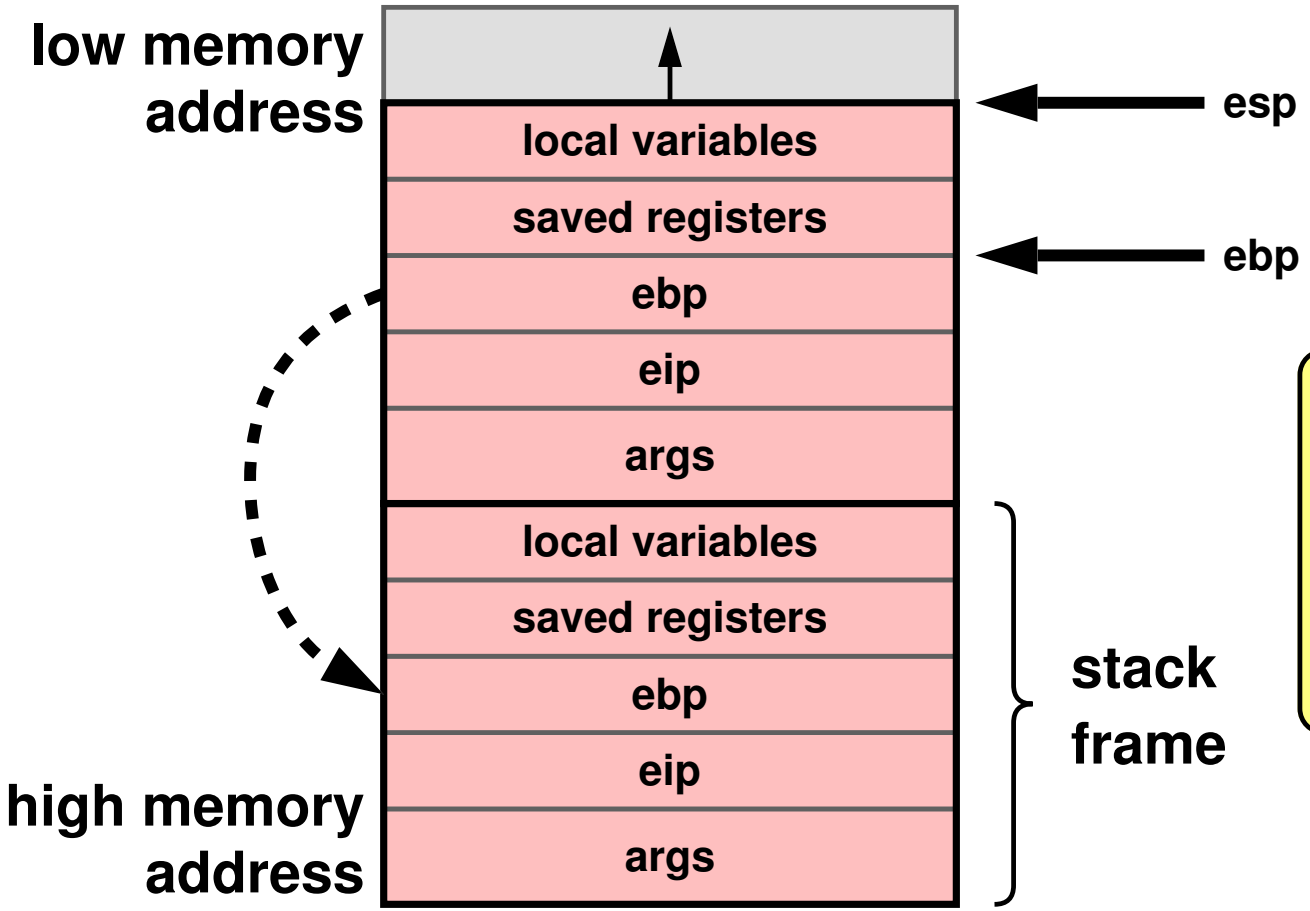
```
int main() {
  int i;
  int a;
  ...
  i = sub(a, 1);
  ...
  return(0);
}
```

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

- The context of `main()` includes CPU registers, any global variables (none here) and its local variables, `i` and `a`
- The context of `sub()` includes
  - any global variables, none here
  - its local variables, `i` and `result`
  - its arguments, `x` and `y`
- Global variables are in fixed location in the address space
- *Local variables* and *arguments* are in *current stack frame*



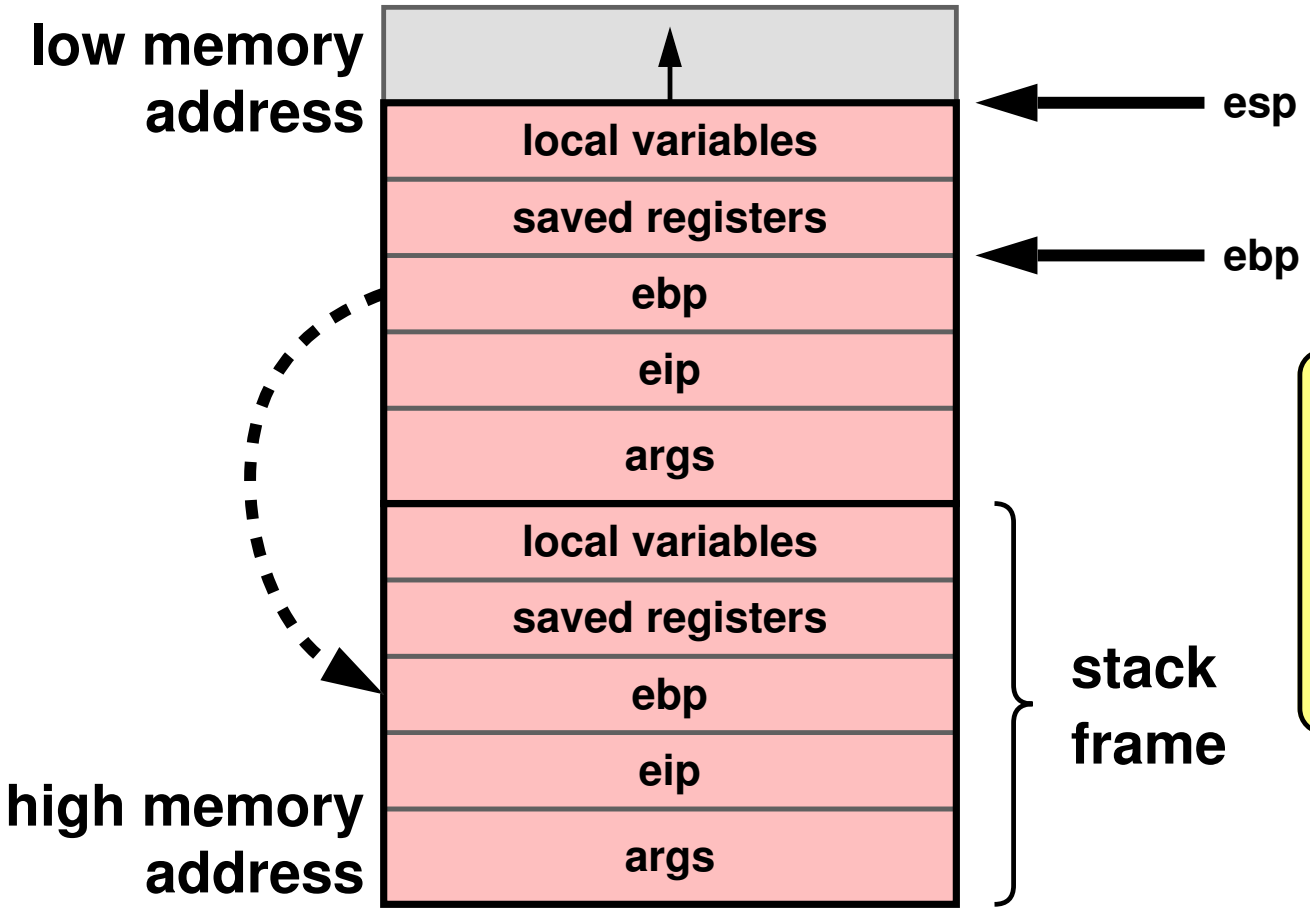
Intel x86 (32-Bit): Subroutine Linkage



- please be reminded that our address space is up-side-down (comparing against the textbook)

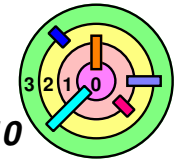
- *esp* points to the end of the current stack frame
  - it is used to prepare the next stack frame
- *eip* contains the caller's *instruction pointer* register
  - this is the *return address*!

Intel x86 (32-Bit): Subroutine Linkage



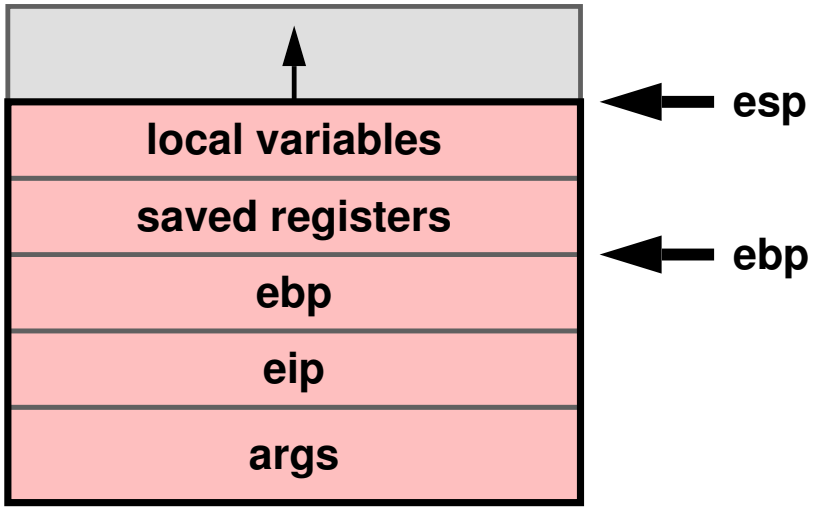
- please be reminded that our address space is up-side-down (comparing against the textbook)

- *ebp* contains the caller's *base (frame) pointer* register
  - this is a link to the caller's *stack frame*
- *eax* contains the return value of a function
- some fields are not alwasy present, compiler decides



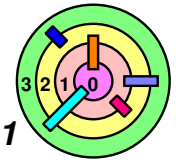
Intel x86 (32-Bit): Subroutine Linkage

- Who sets what?
  - *args* is explicitly setup by the *caller*
  - *eip* is copied into the stack frame by a "`call`" machine instruction in the *caller* function
  - *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



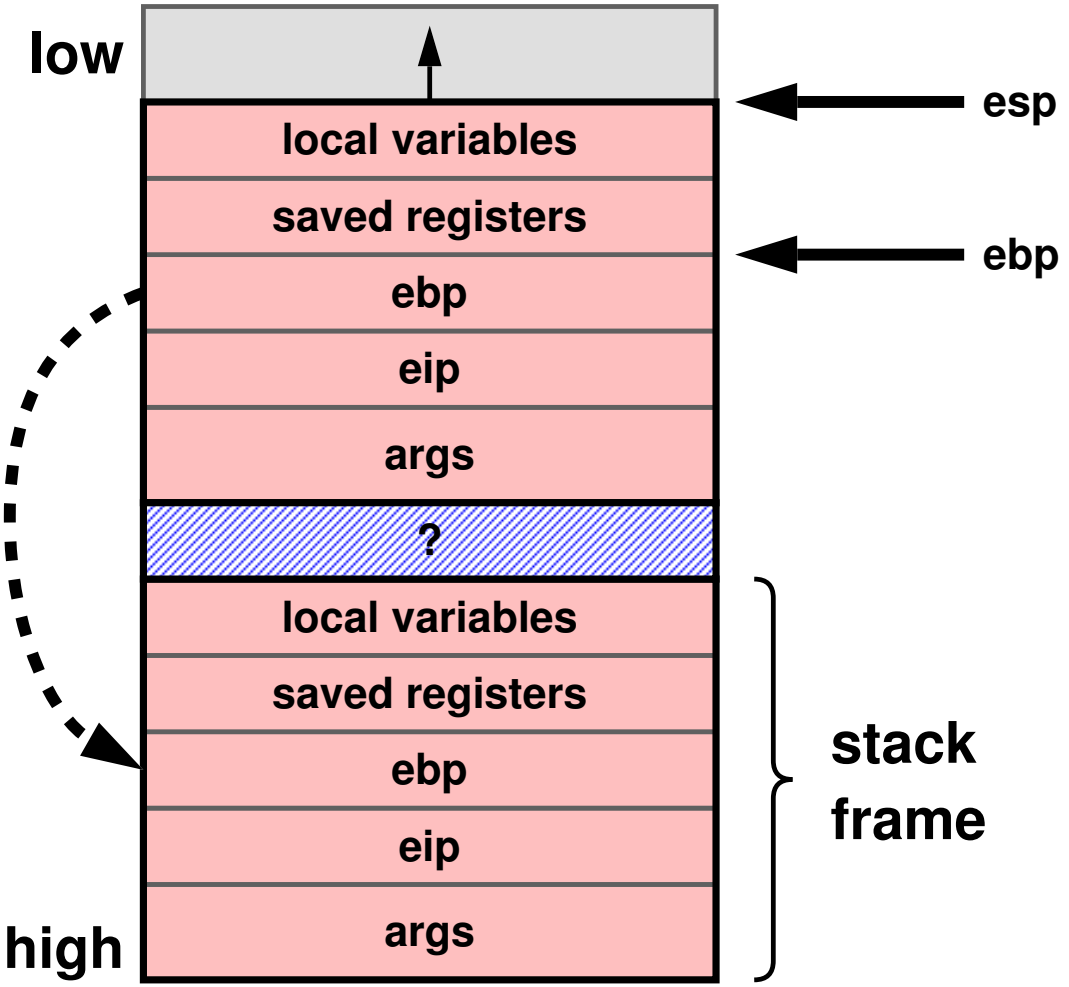
- What does the stack frame look like for the following function?

```
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)

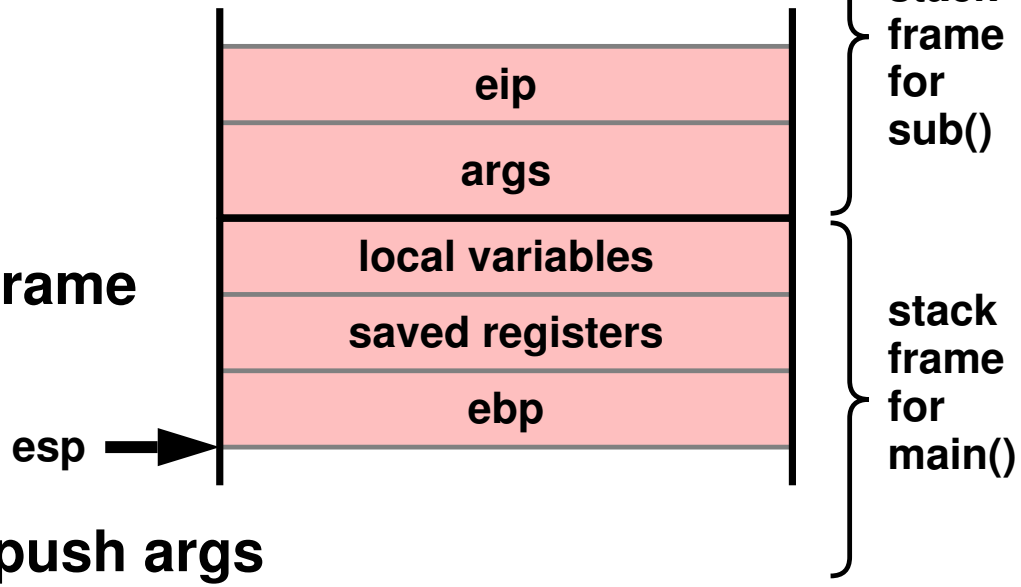
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

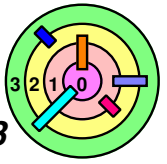
push args

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
```

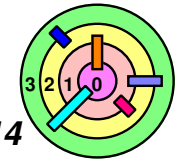
set up stack frame

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)

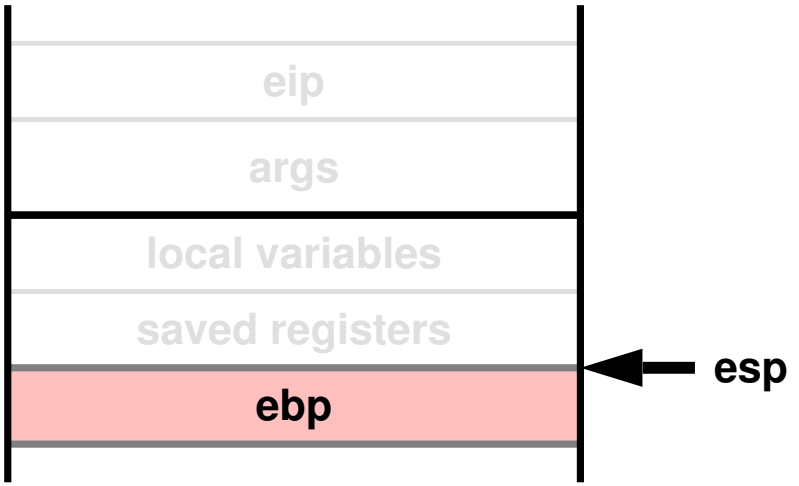
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

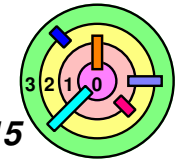
push args

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)

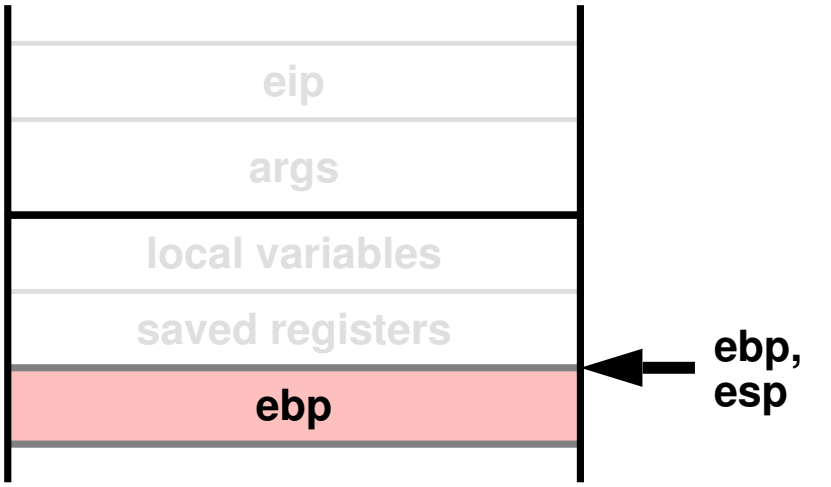
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

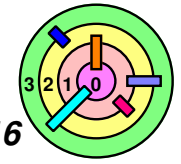
push args

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)

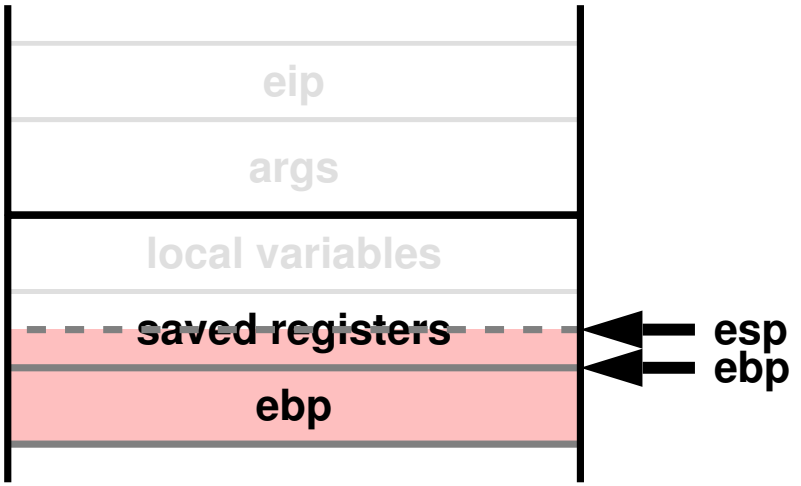
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

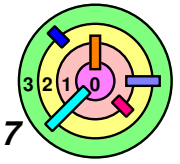
push args

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)

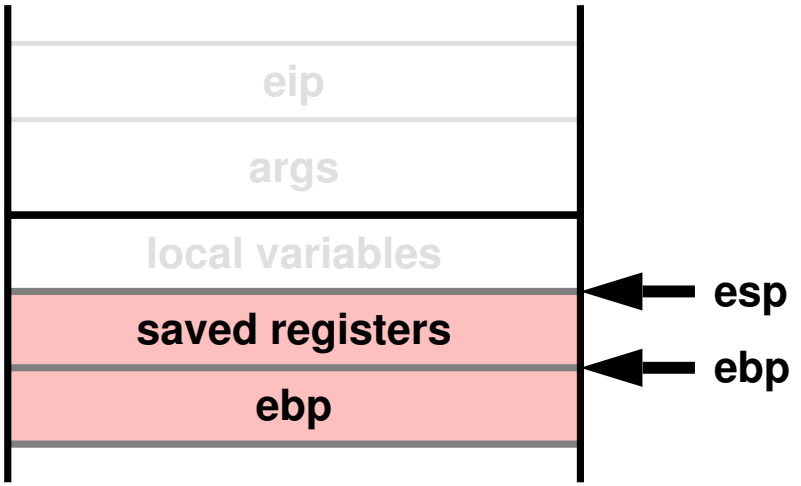
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

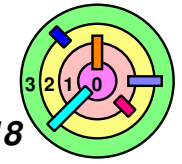
push args

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)

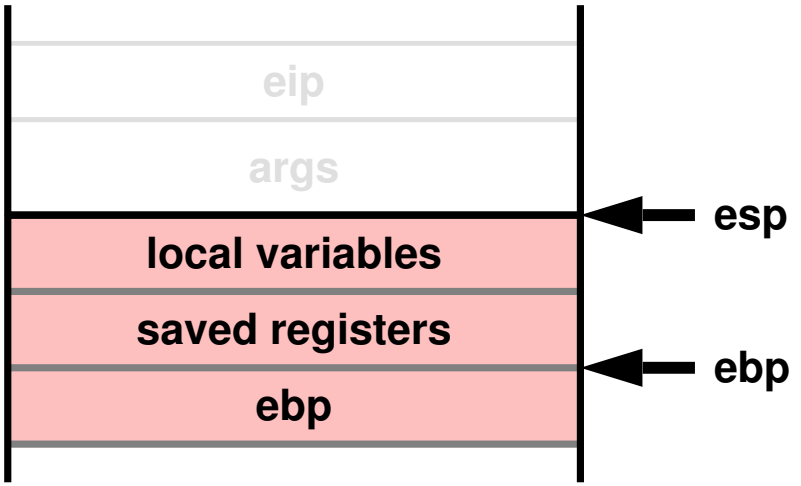
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

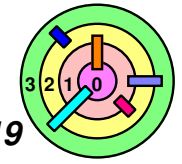
push args

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)

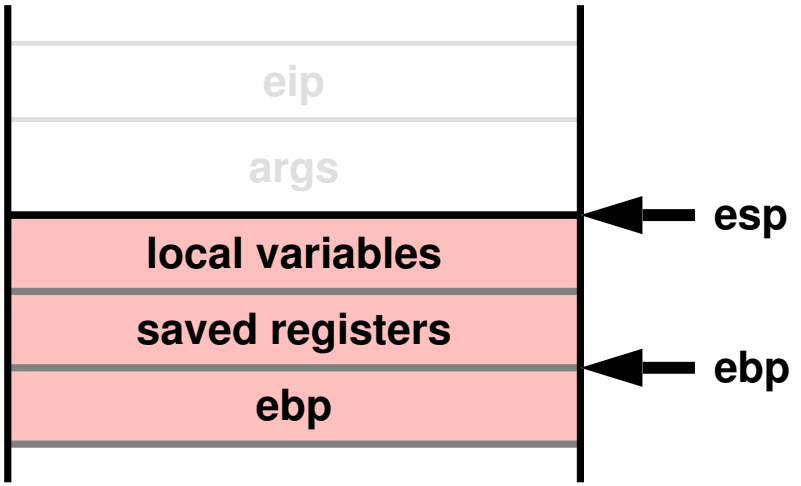
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

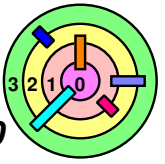
push args

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)

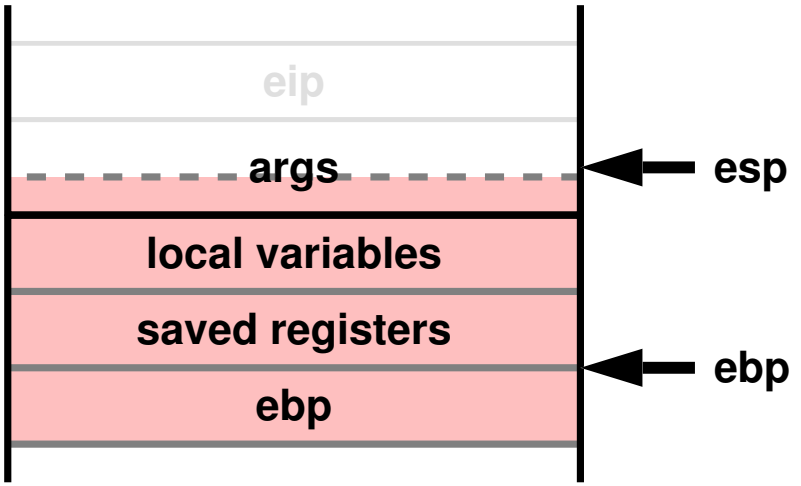
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

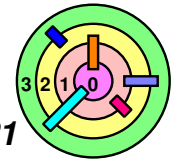
push args

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)

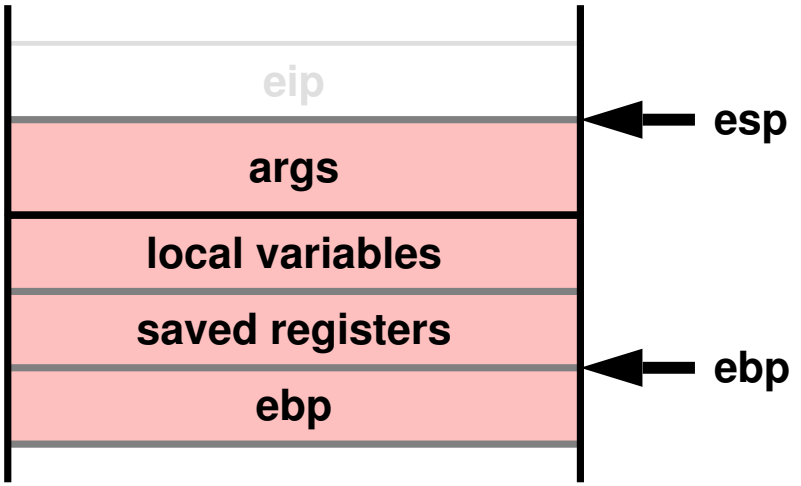
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

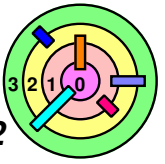
push args

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)

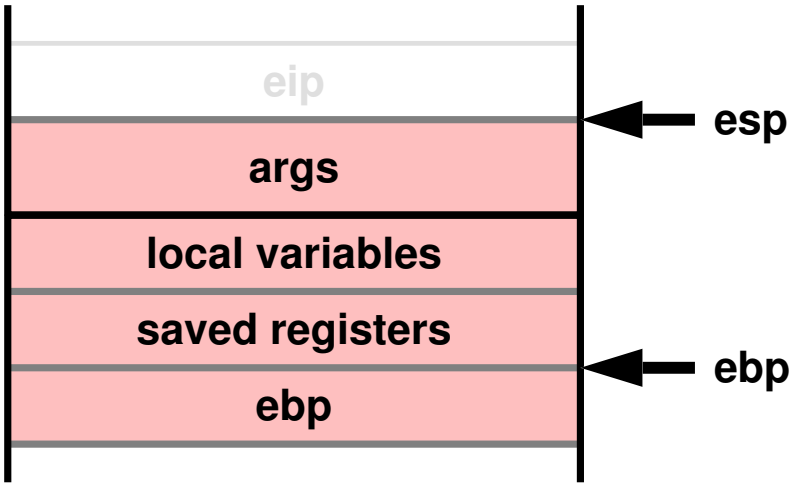
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

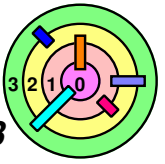
push args

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)

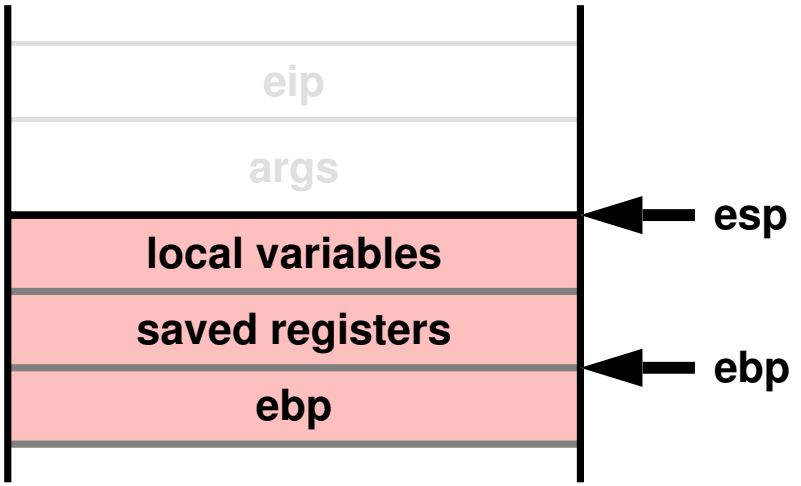
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

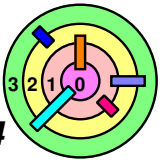
push args

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)

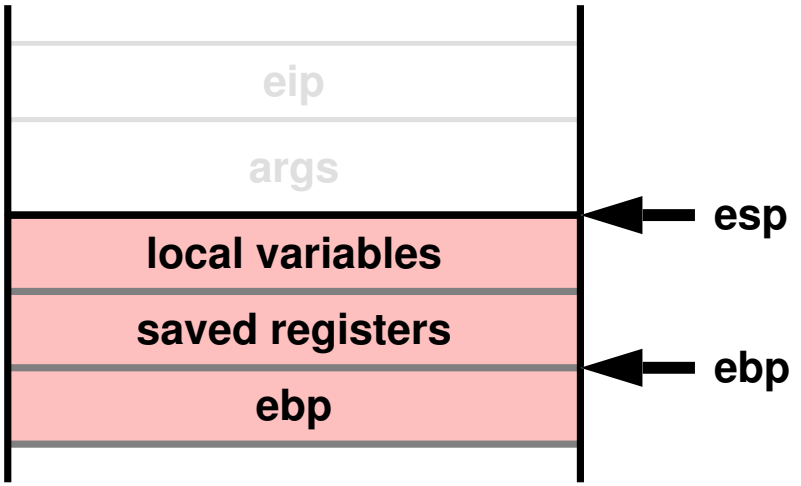
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

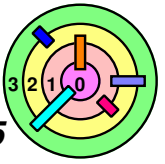
push args

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)

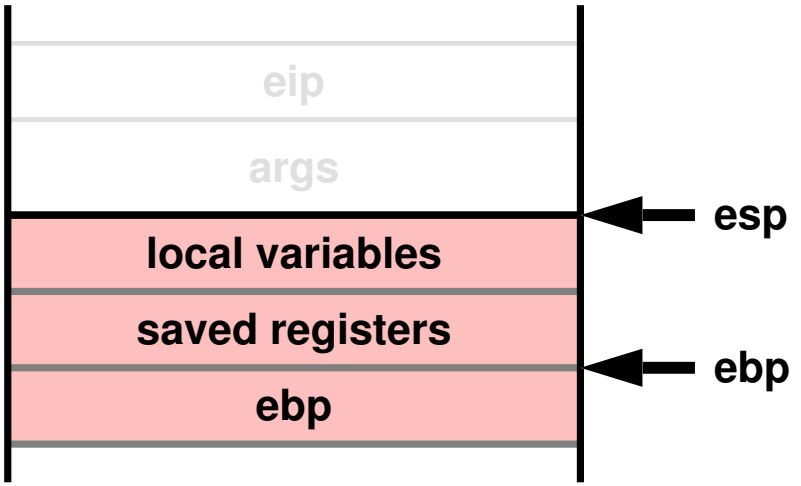
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

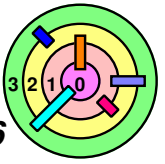
push args

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)

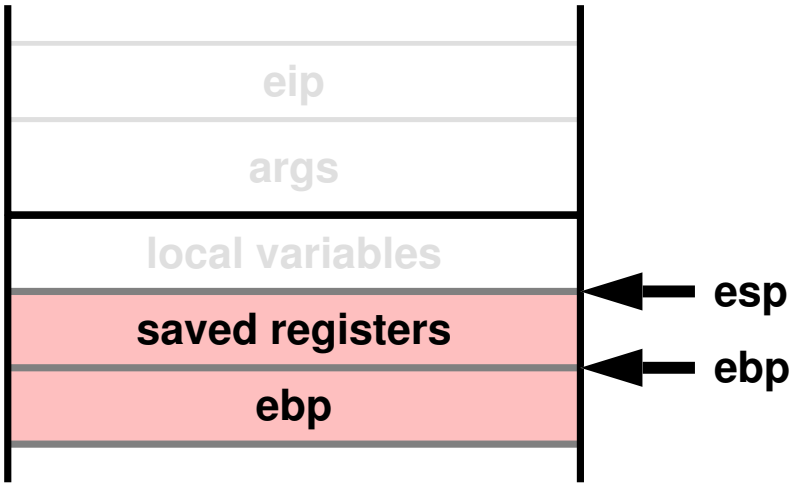
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

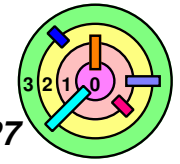
push args

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)

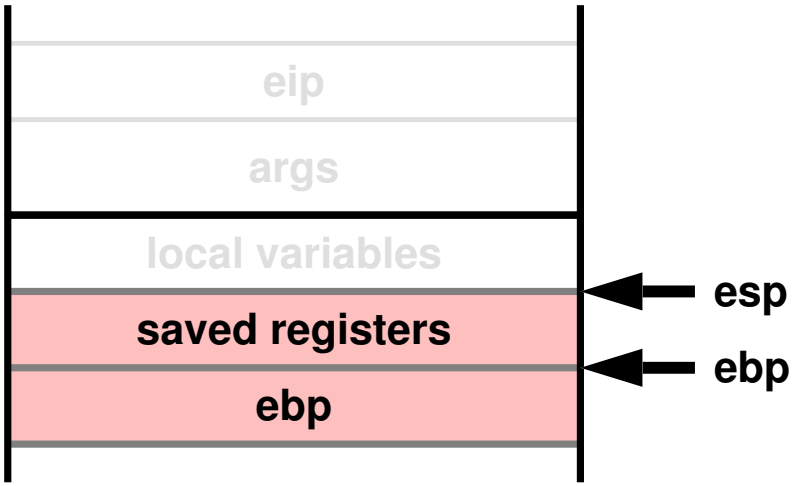
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

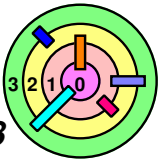
push args

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)

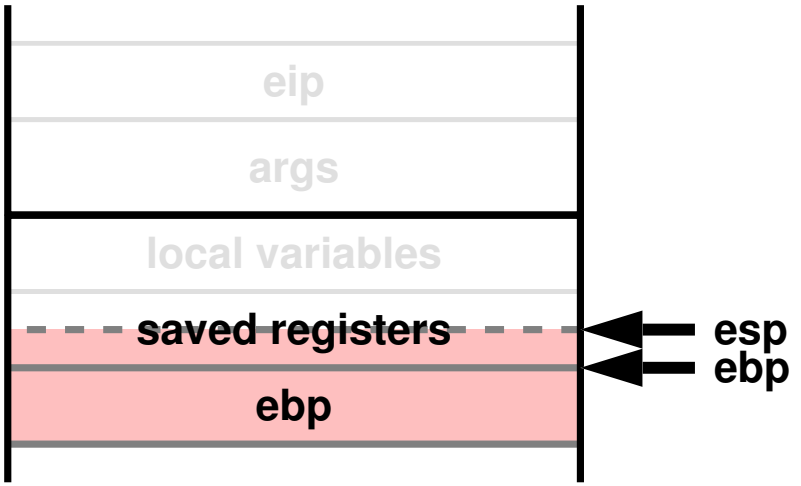
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

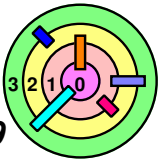
push args

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)

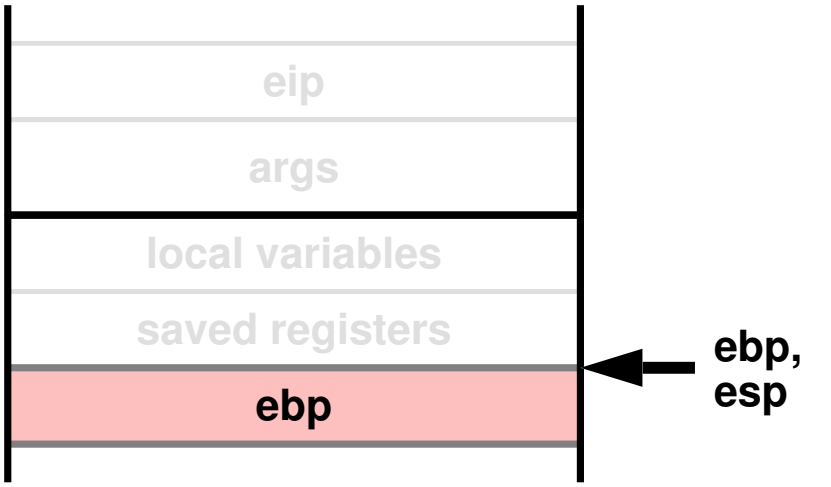
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

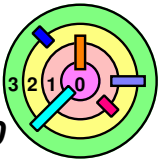
push args

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)

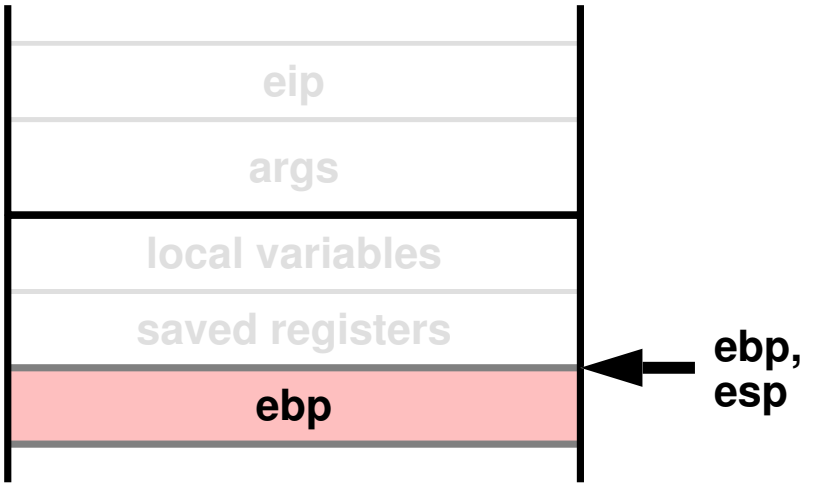
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

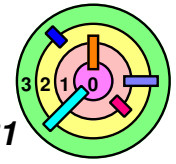
push args

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
```

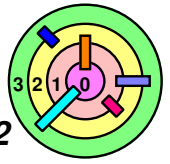
set up stack frame

push args

pop args; get result

set return value and restore frame

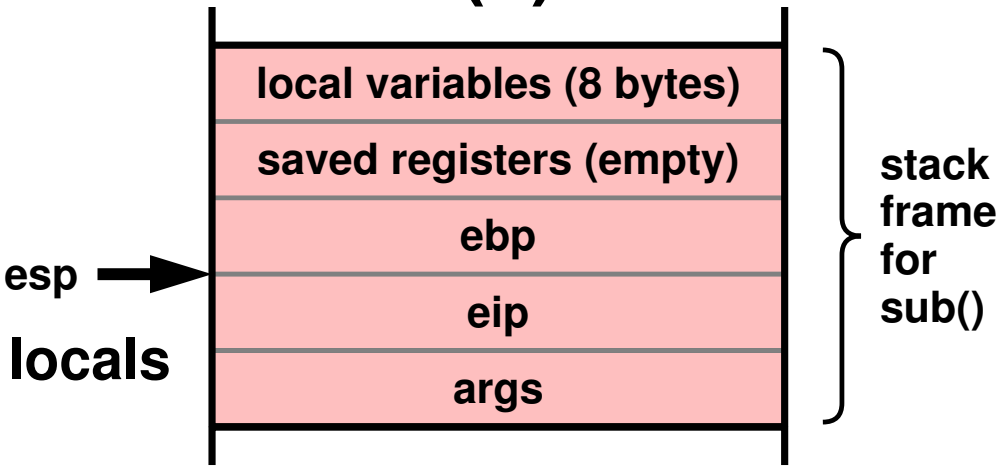

```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



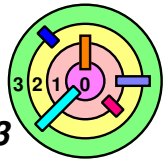
Intel x86: Subroutine Code (2)

```
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



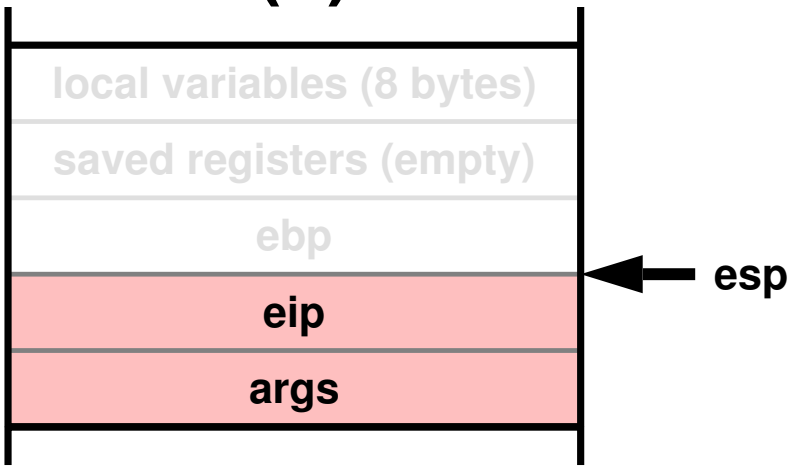

```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```



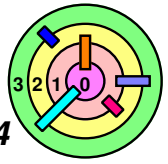
Intel x86: Subroutine Code (2)

```
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



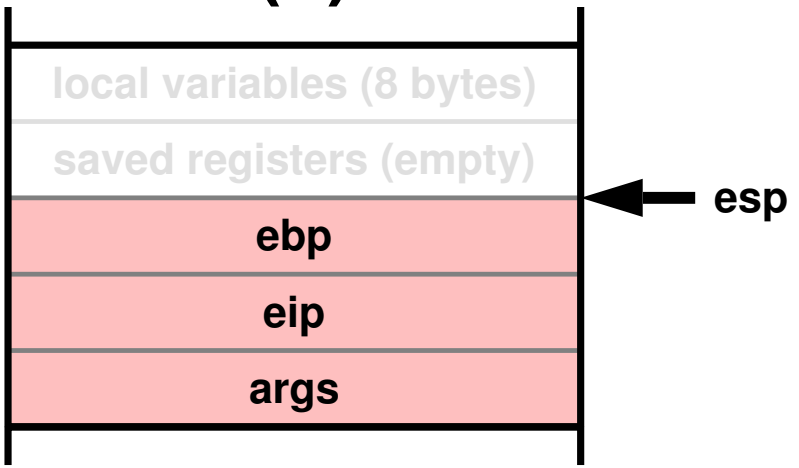

```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```



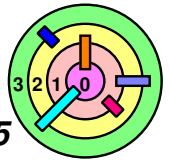
Intel x86: Subroutine Code (2)

```
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



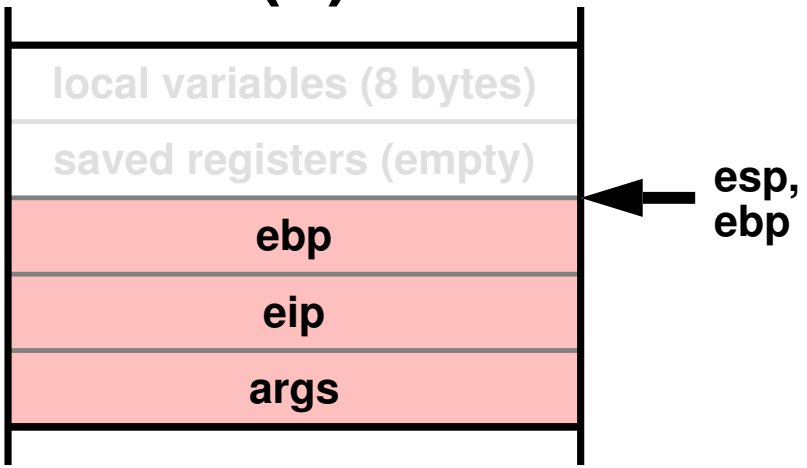

```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```



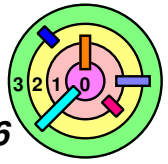
Intel x86: Subroutine Code (2)

```
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



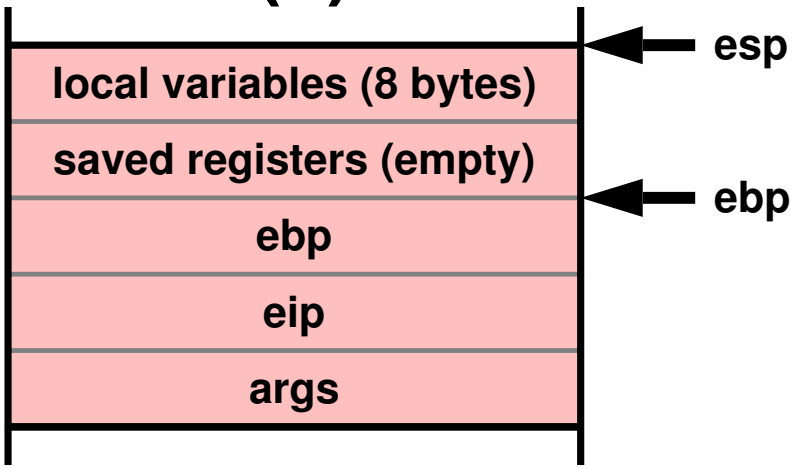

```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```



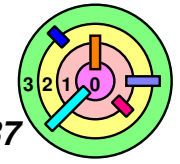
Intel x86: Subroutine Code (2)

```
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



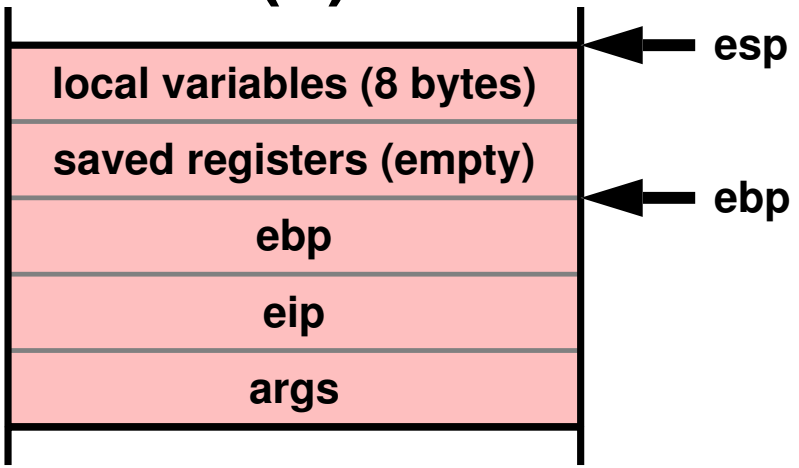

```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```



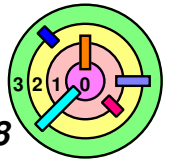
Intel x86: Subroutine Code (2)

```
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



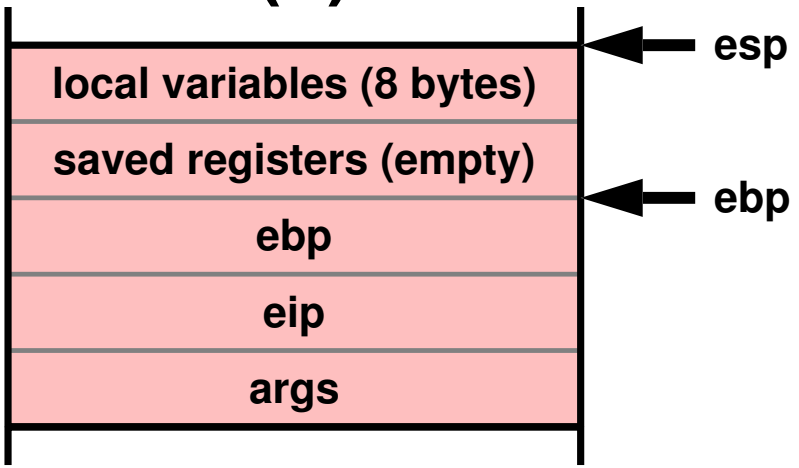

```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```



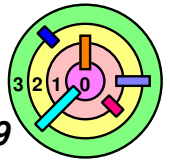
Intel x86: Subroutine Code (2)

```
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



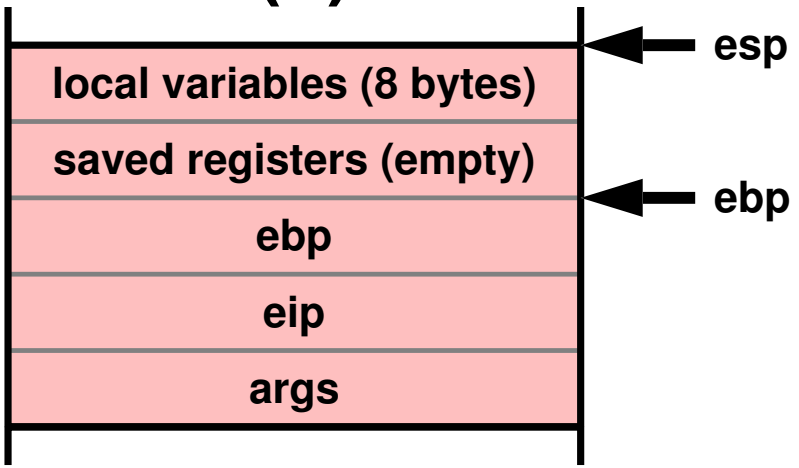

```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```



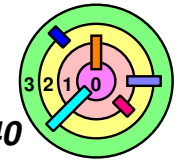
Intel x86: Subroutine Code (2)

```
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



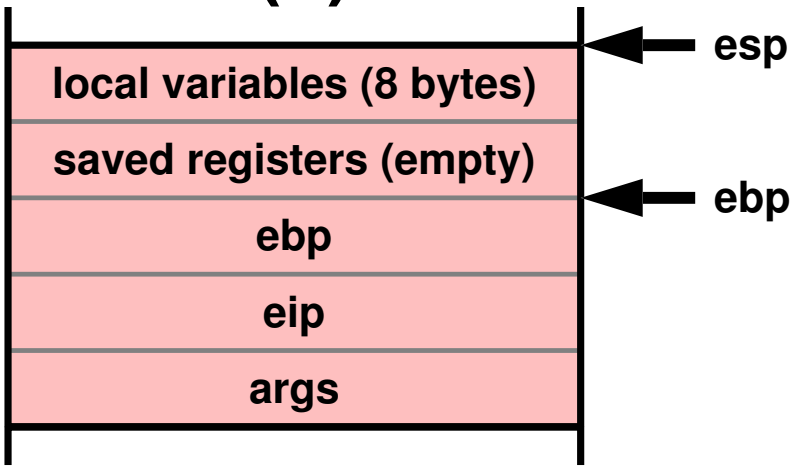

```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```



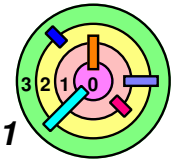
Intel x86: Subroutine Code (2)

```
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



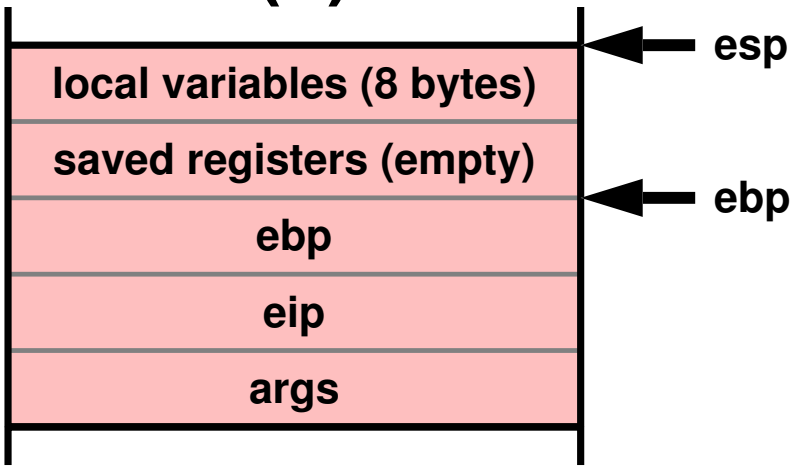

```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```



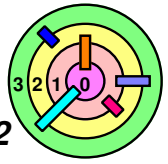
Intel x86: Subroutine Code (2)

```
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



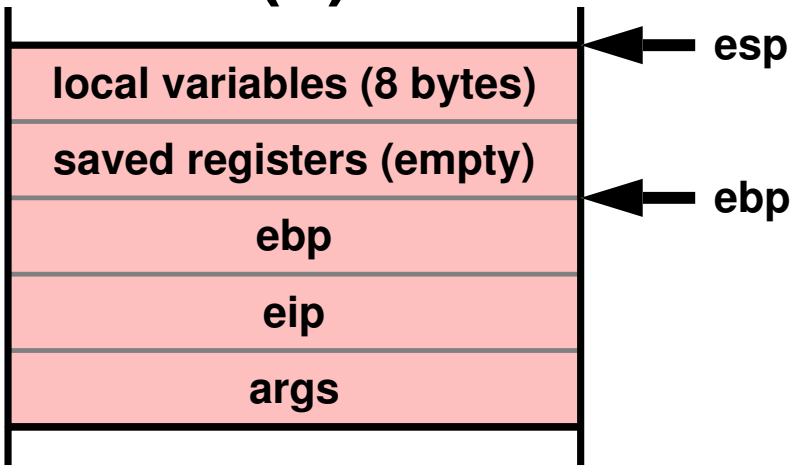

```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```



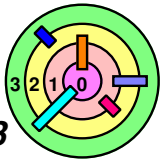
Intel x86: Subroutine Code (2)

```
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



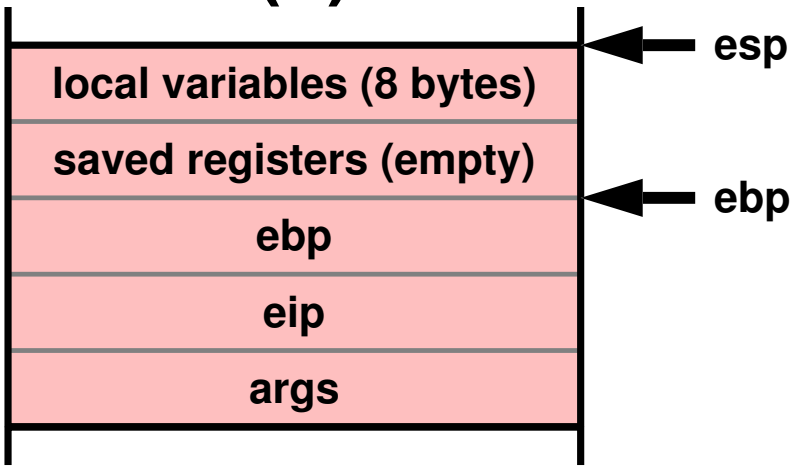

```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```



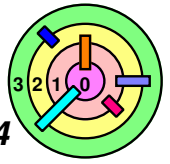
Intel x86: Subroutine Code (2)

```
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



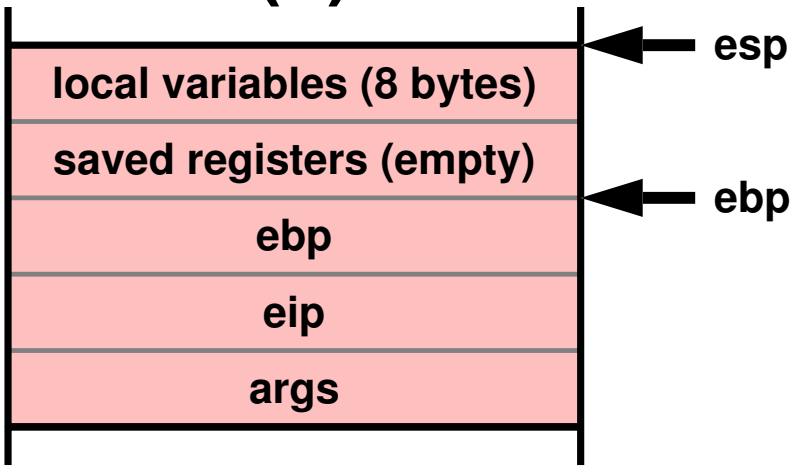

```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```



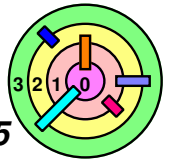
Intel x86: Subroutine Code (2)

```
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



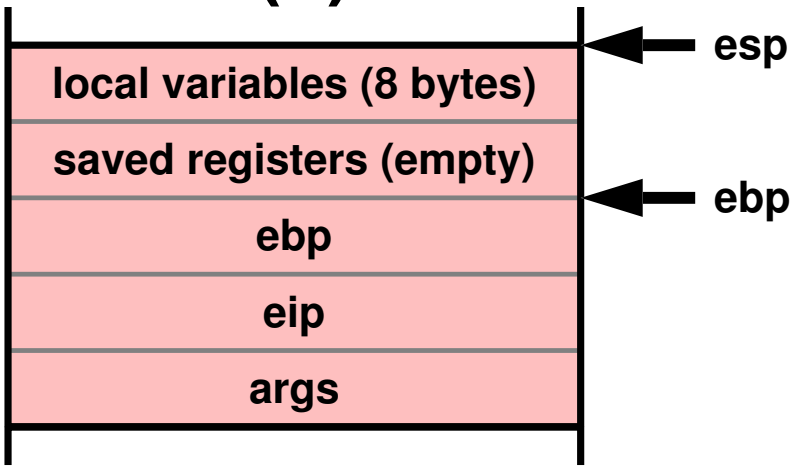

```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```



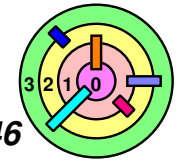
Intel x86: Subroutine Code (2)

```
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



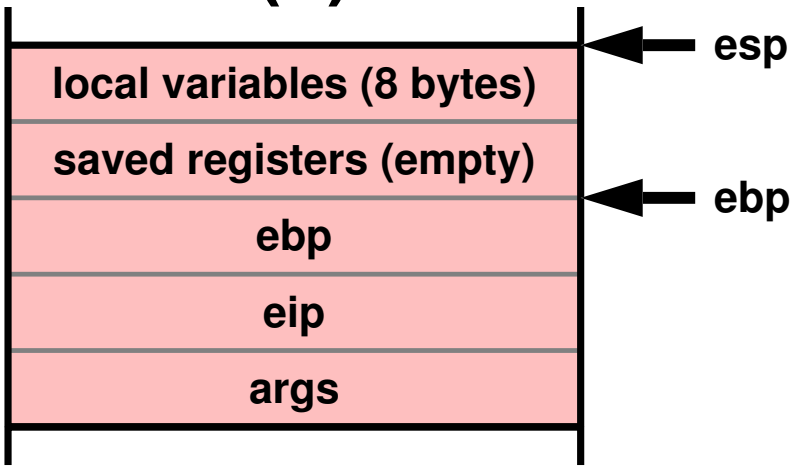

```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```



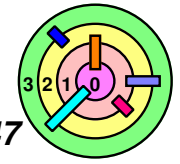
Intel x86: Subroutine Code (2)

```
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



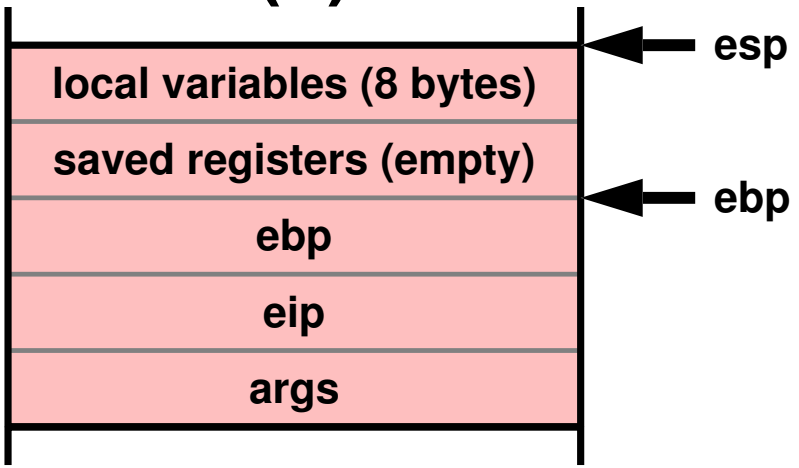

```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```



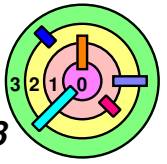
Intel x86: Subroutine Code (2)

```
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



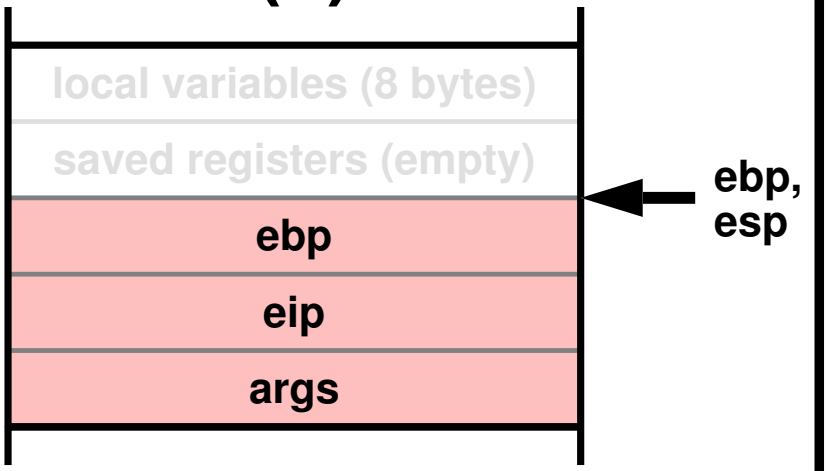

```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```



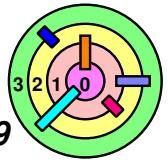
Intel x86: Subroutine Code (2)

```
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



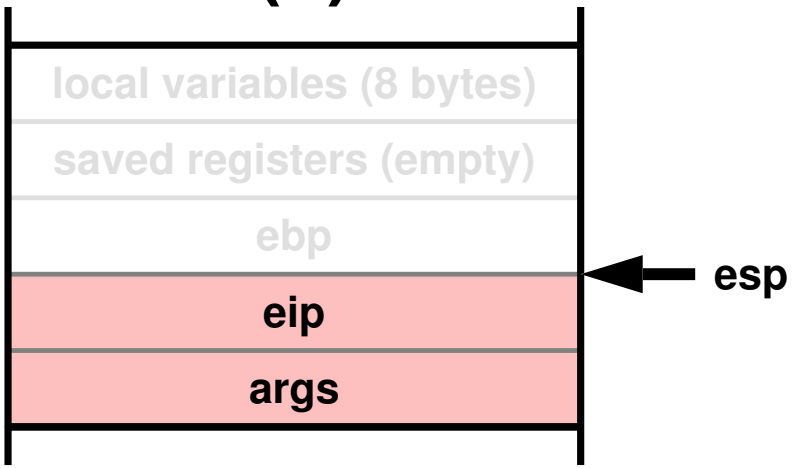

```
int sub(int x, int y) {
   // computers x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```



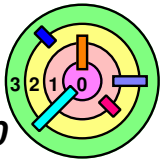
Intel x86: Subroutine Code (2)

```
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




```
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

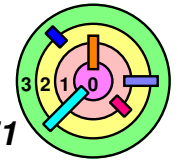
Output Registers

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

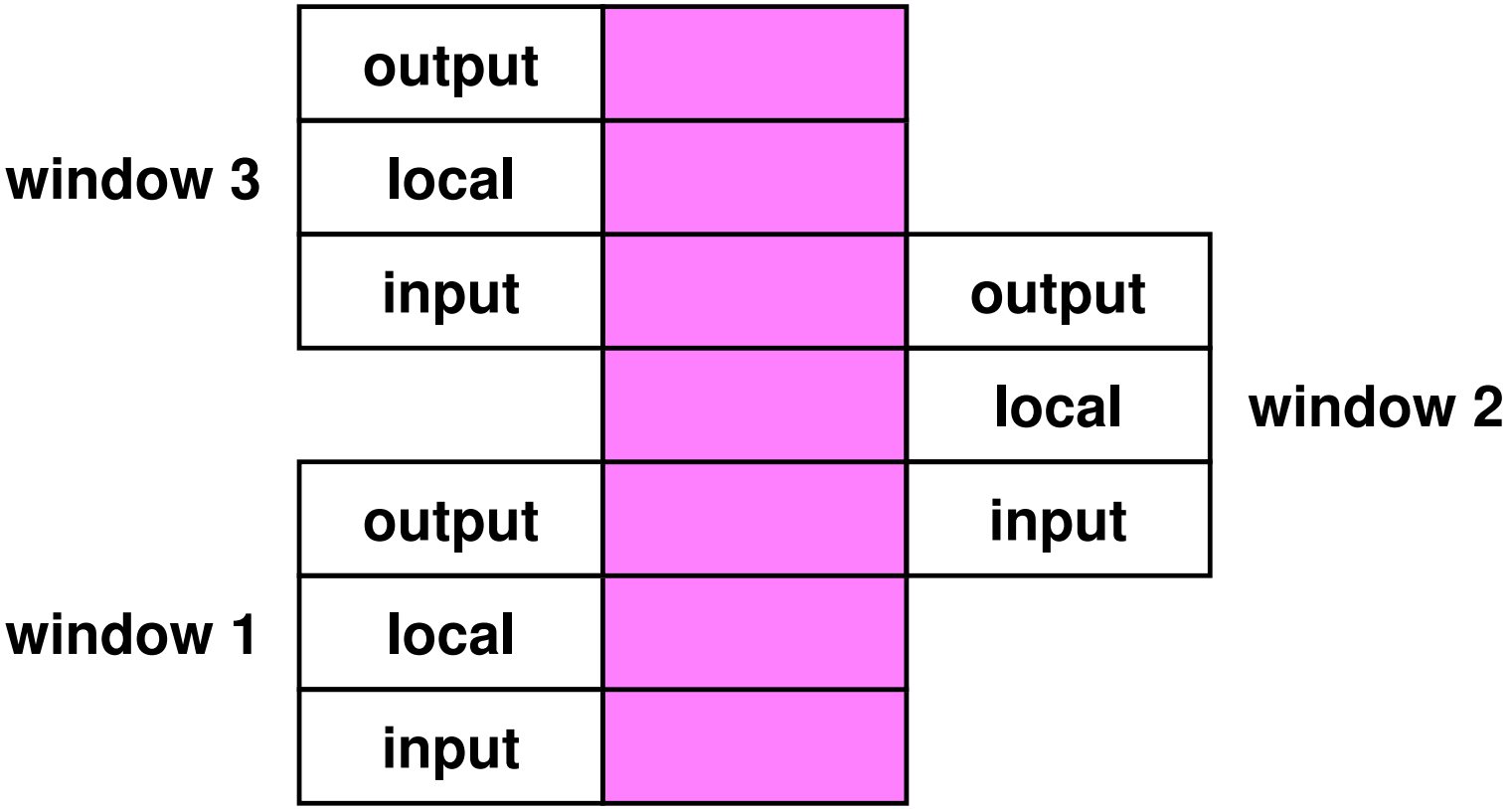
Local Registers

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

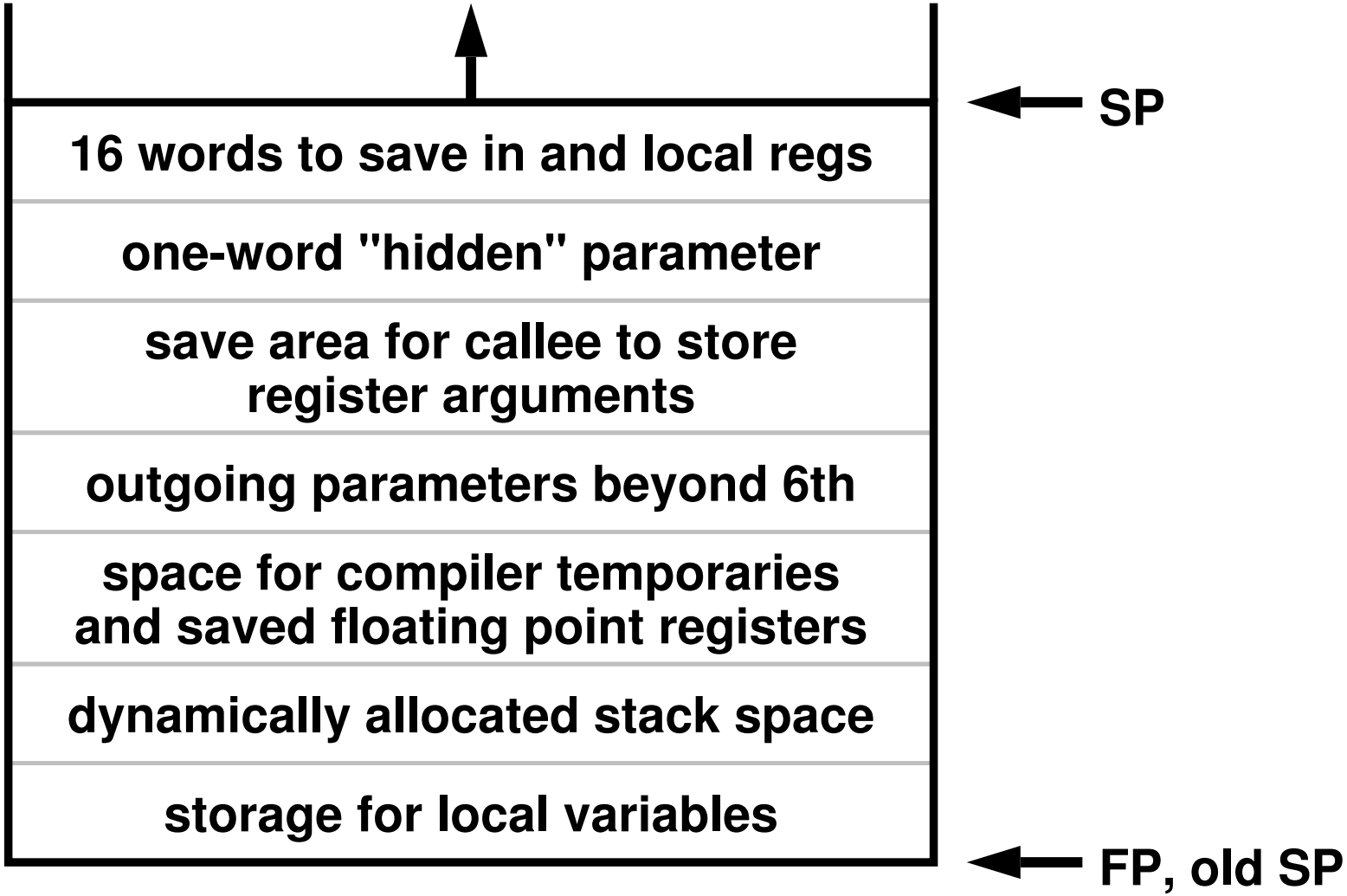
Global Registers



SPARC Architecture: Register Windows

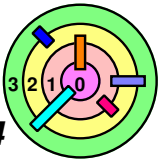


SPARC Architecture: Stack



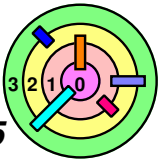
SPARC Architecture: Subroutine Code

```
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)
```

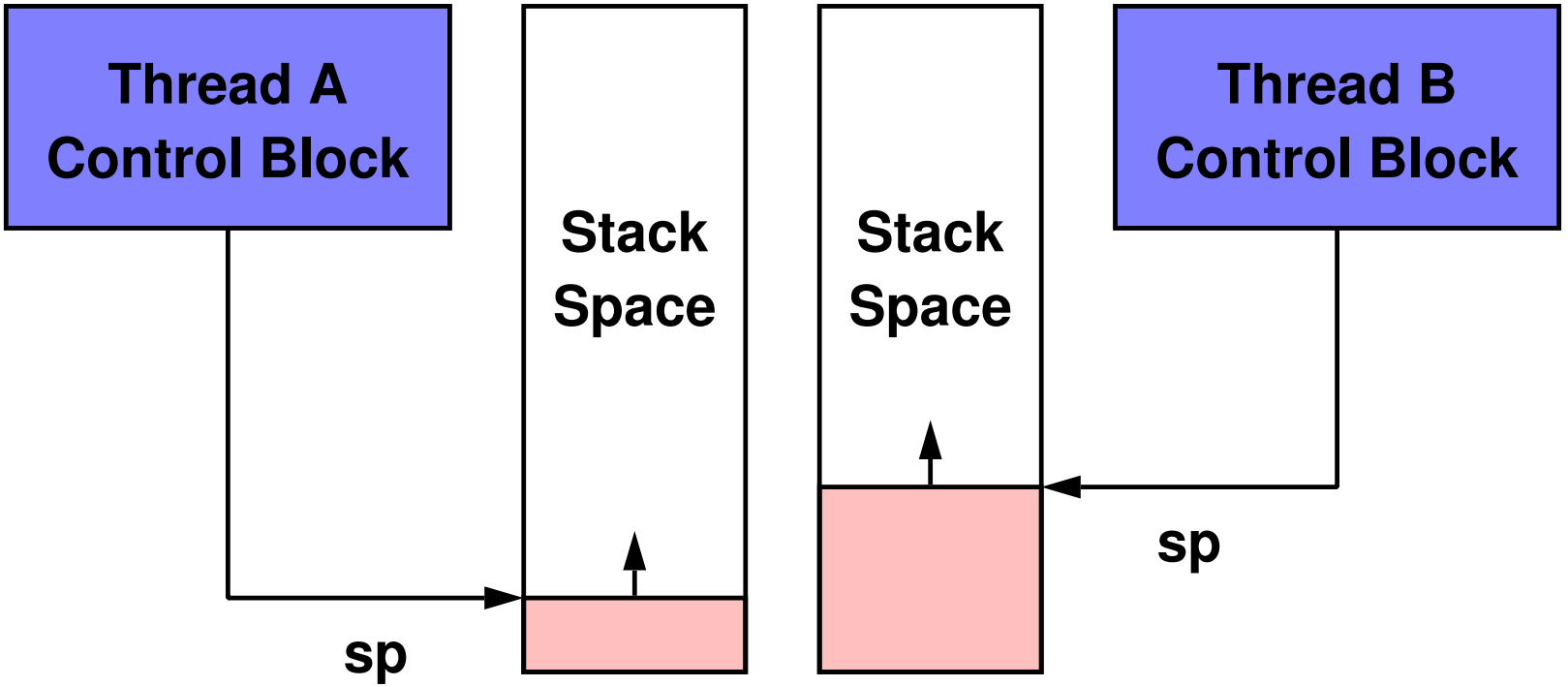


3.1 Context Switching

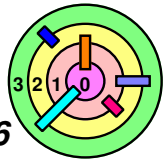
- Procedures
- *Threads & Coroutines*
- Systems Calls
- Interrupts



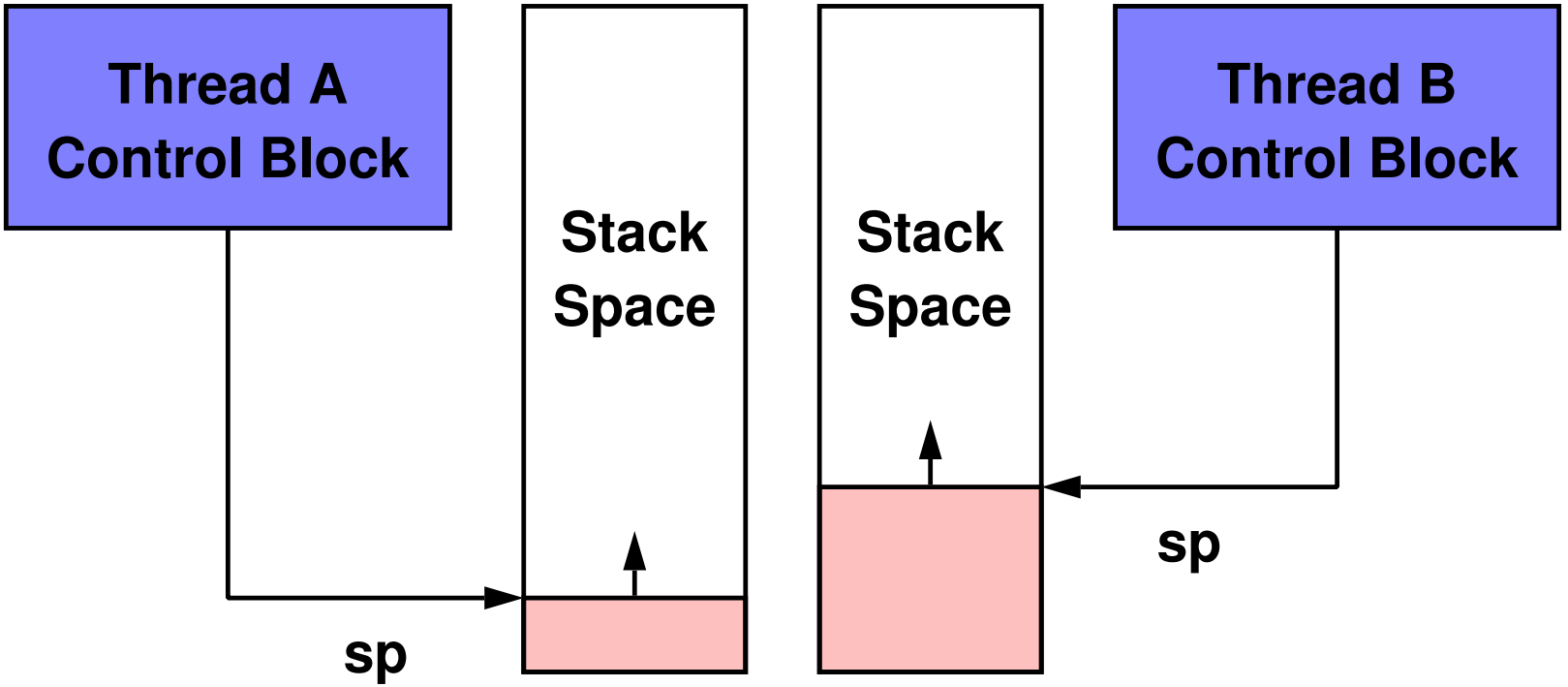
Representing Threads



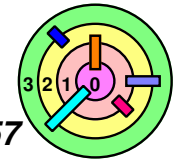
- normally, threads are independent of one another and don't directly control one another's execution
- threads can be made aware of each other and be able to *transfer control* from one thread to another
  - this is known as *coroutine linkage*



Representing Threads

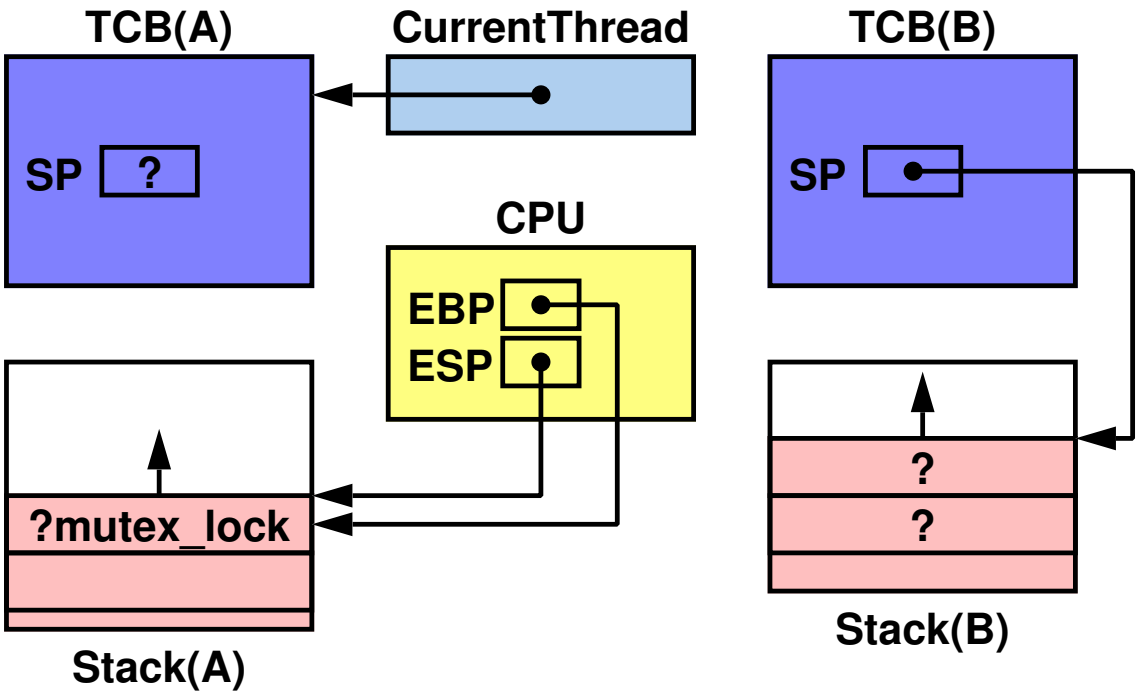


- A thread's context
  - its stack, its register state
  - can be stored in a *thread control block* (directly or indirectly)
- To transfer control from one thread to another is equivalent to copying the thread control block of the target thread into the current thread context

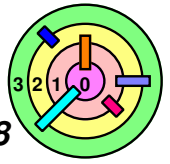


Switching Between Threads

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



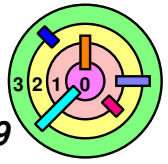
- thread A calls `switch()` to switch to thread B
- thread B's TCB has the *exact context* of thread B right before thread B was *last suspended*
- context information in thread A's TCB is *out-dated*



Switching Between Threads

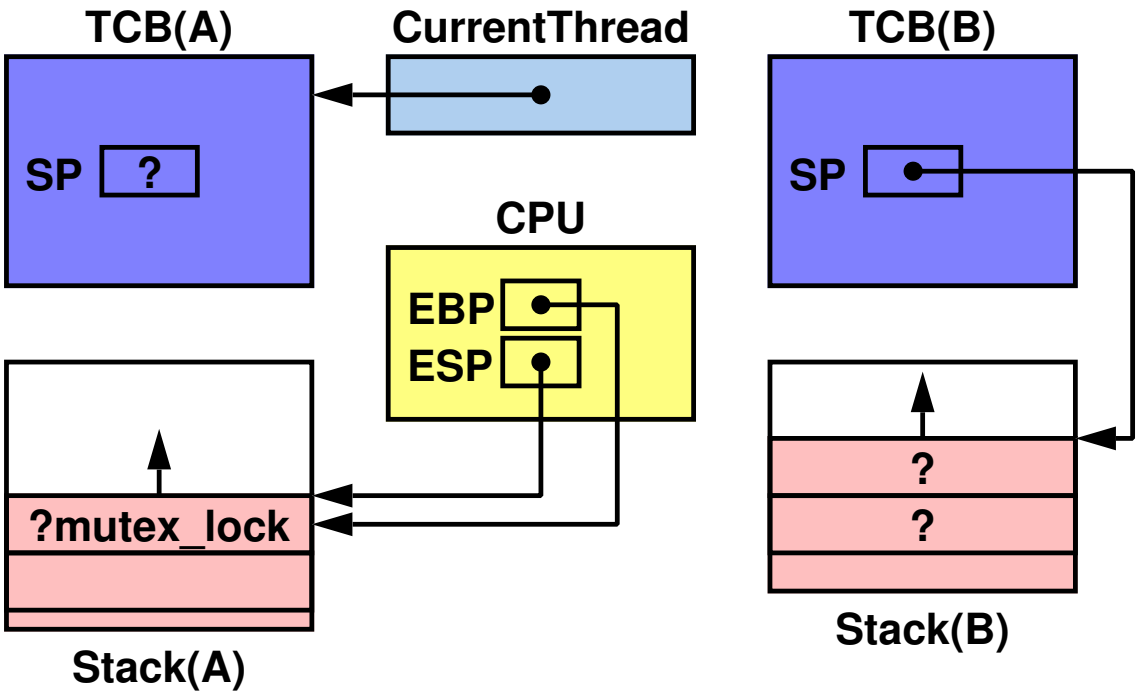
```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```

```
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
```

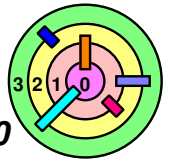


Switching Between Threads

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

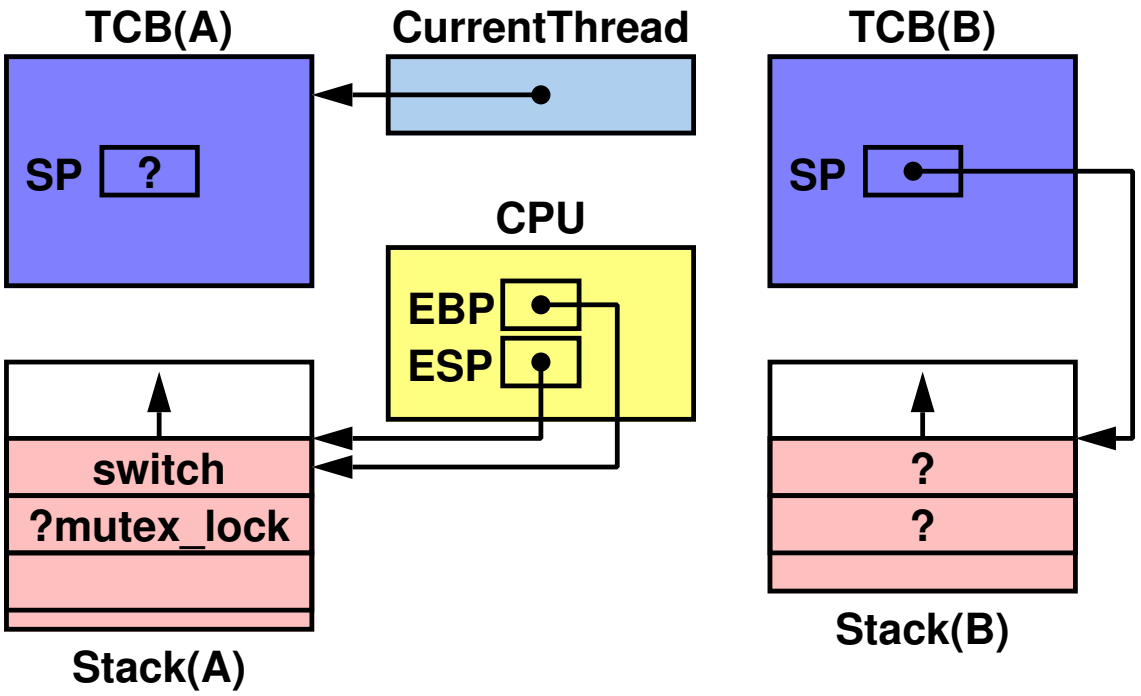


- on entry into `switch()`, the caller's registers are saved!



Switching Between Threads

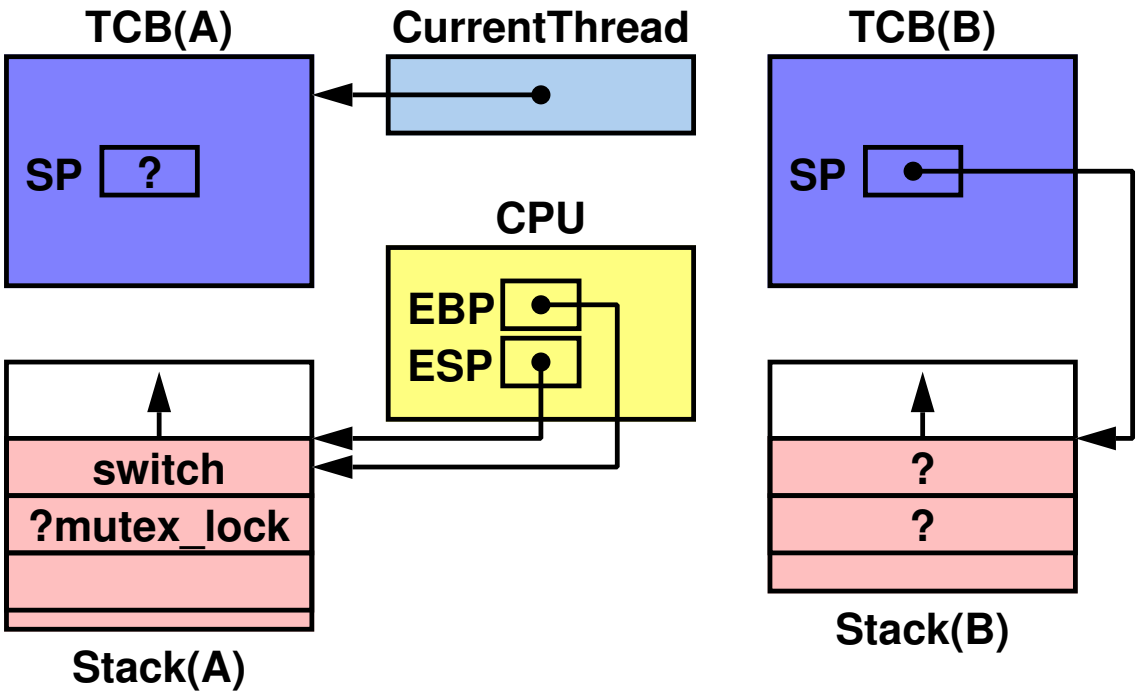
```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```



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

Switching Between Threads

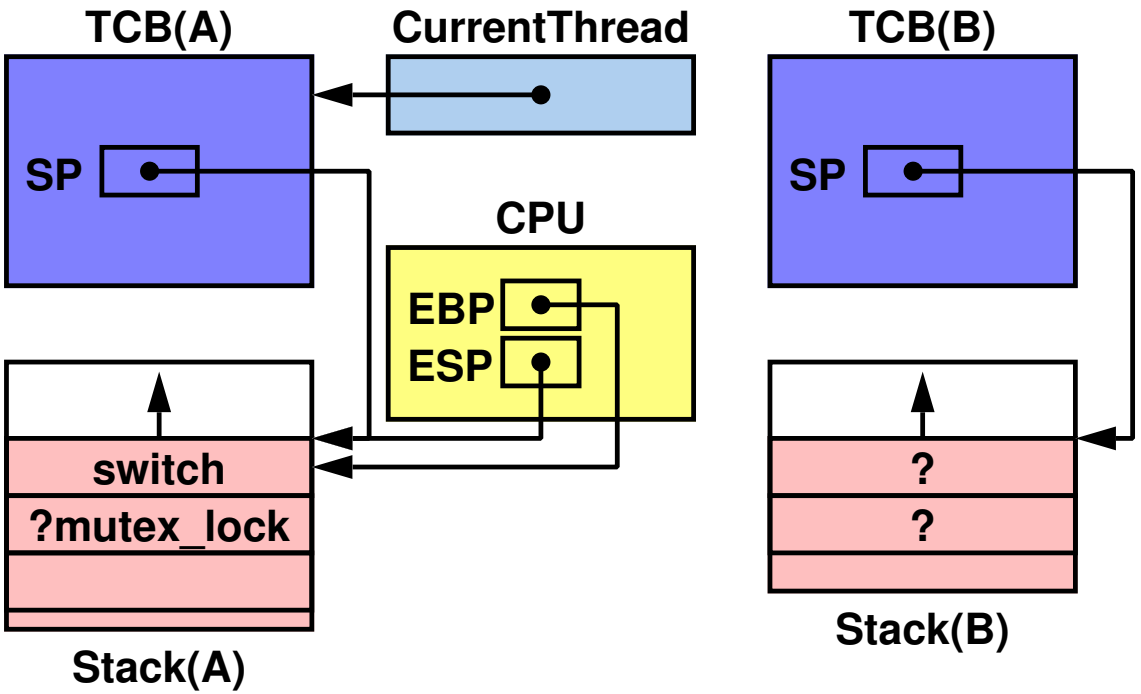
```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```



- then the current stack pointer is saved into current thread's thread control block

Switching Between Threads

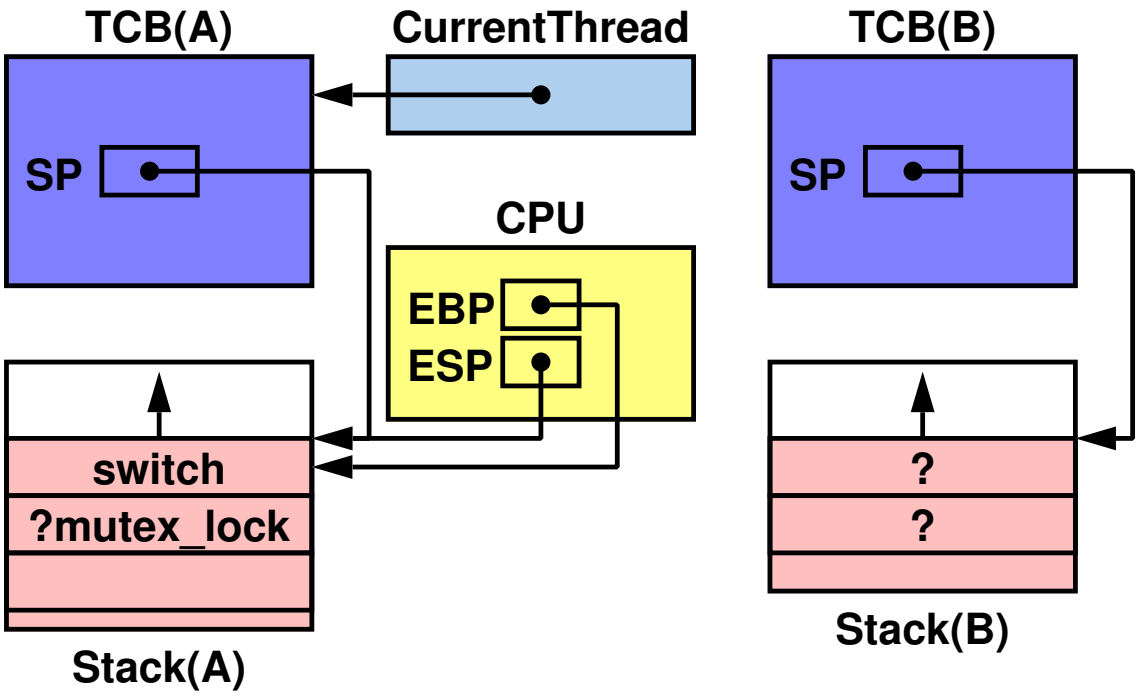
```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```



- then the current stack pointer is saved into current thread's thread control block

Switching Between Threads

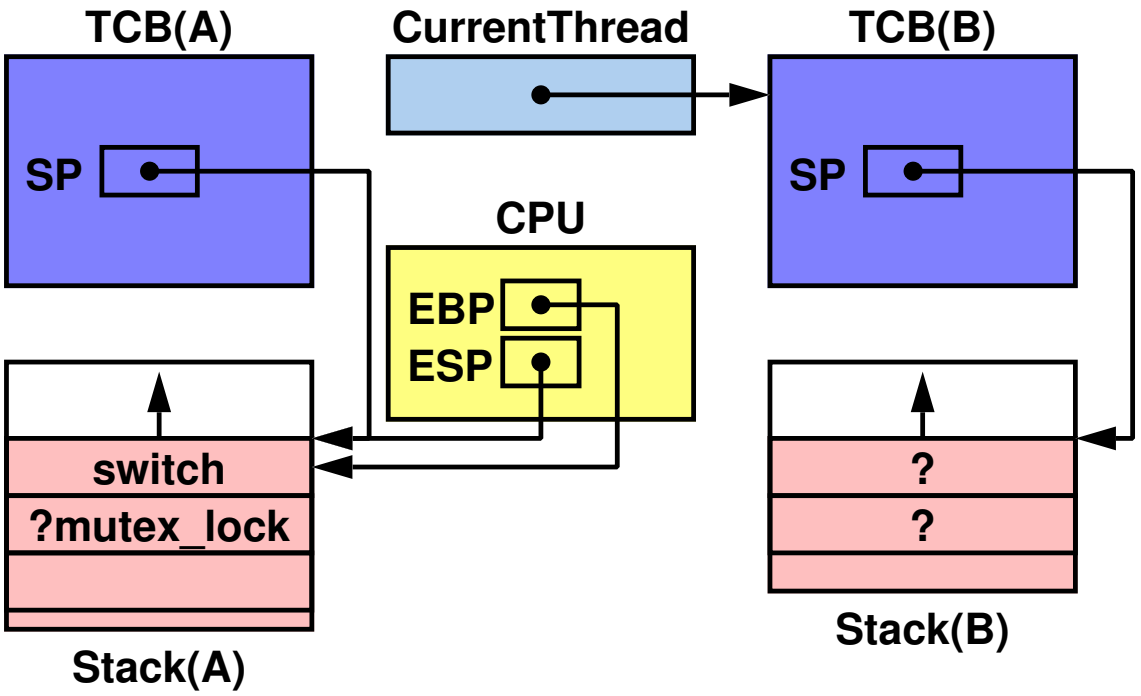
```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```



- the thread control block of the target thread is copied into the current thread context

Switching Between Threads

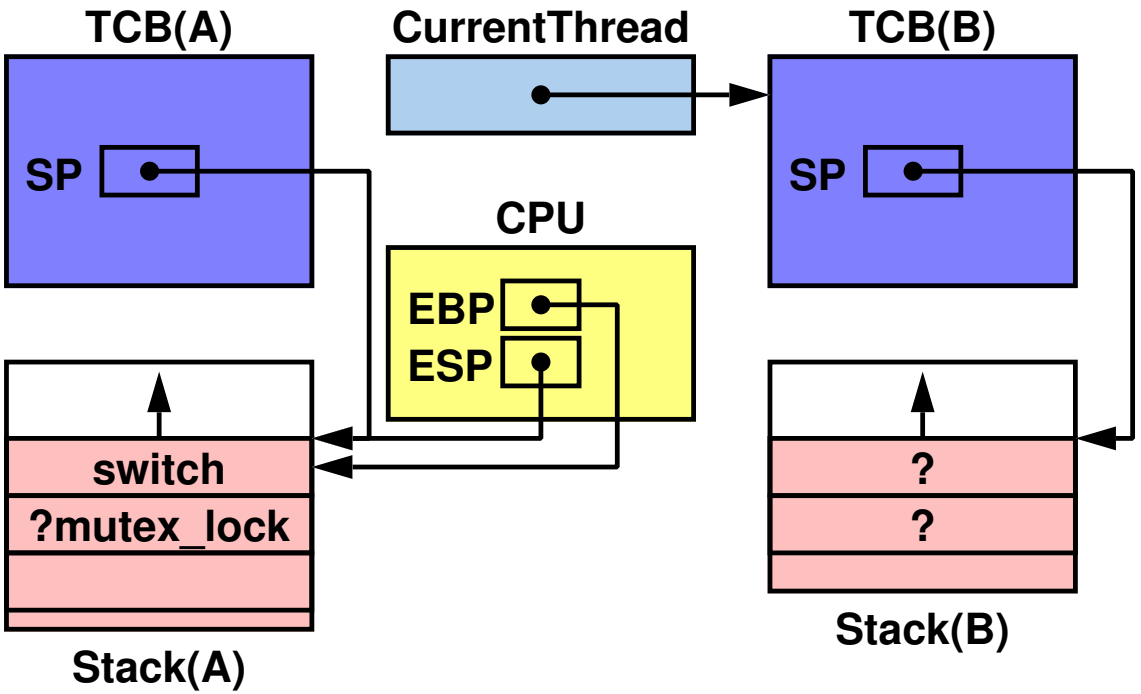
```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```



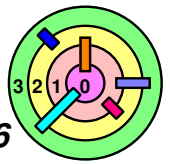
- the thread control block of the target thread is copied into the current thread context

Switching Between Threads

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

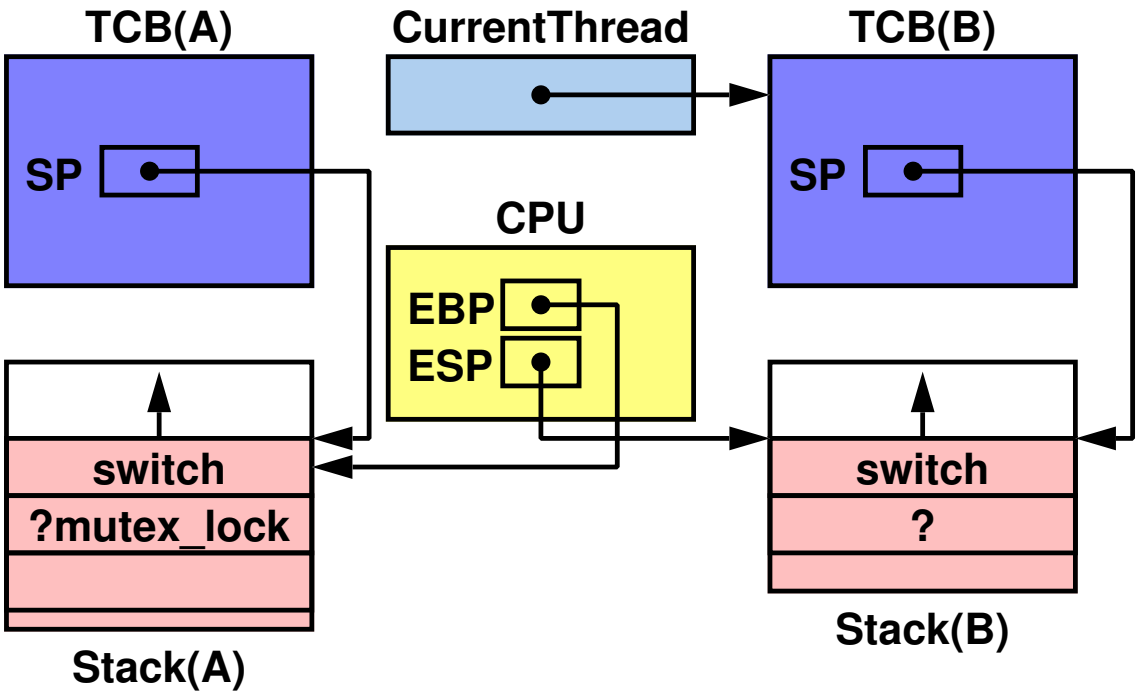


- fetch the target thread's stack pointer (esp for x86) from its thread control block and loads it into the actual stack pointer
  - which thread executes this?
    - does it matter?

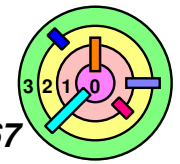


Switching Between Threads

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

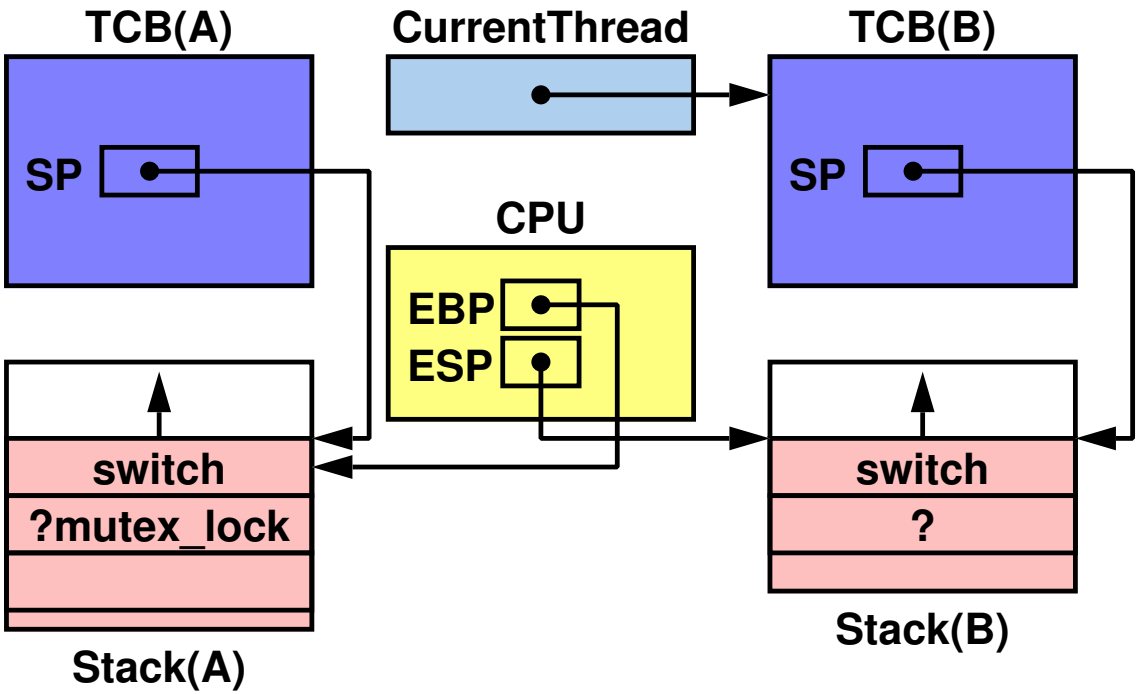


- fetch the target thread's stack pointer (esp for x86) from its thread control block and loads it into the actual stack pointer
  - hmm... which thread executes this?
    - both? either? EIP?



Switching Between Threads

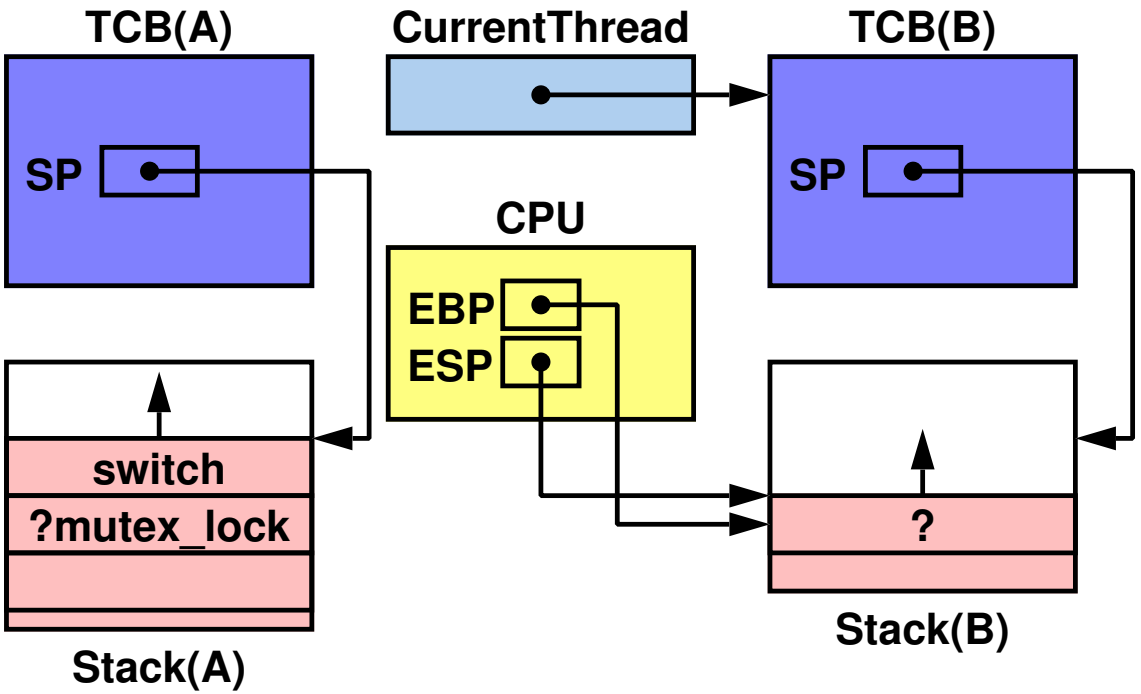
```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```



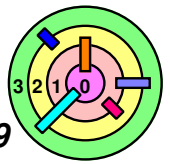
- on return from `switch()`, the registers (ebp and eip for x86) are restored into the current thread, which is the target thread!
  - which thread executes `return`?

Switching Between Threads

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

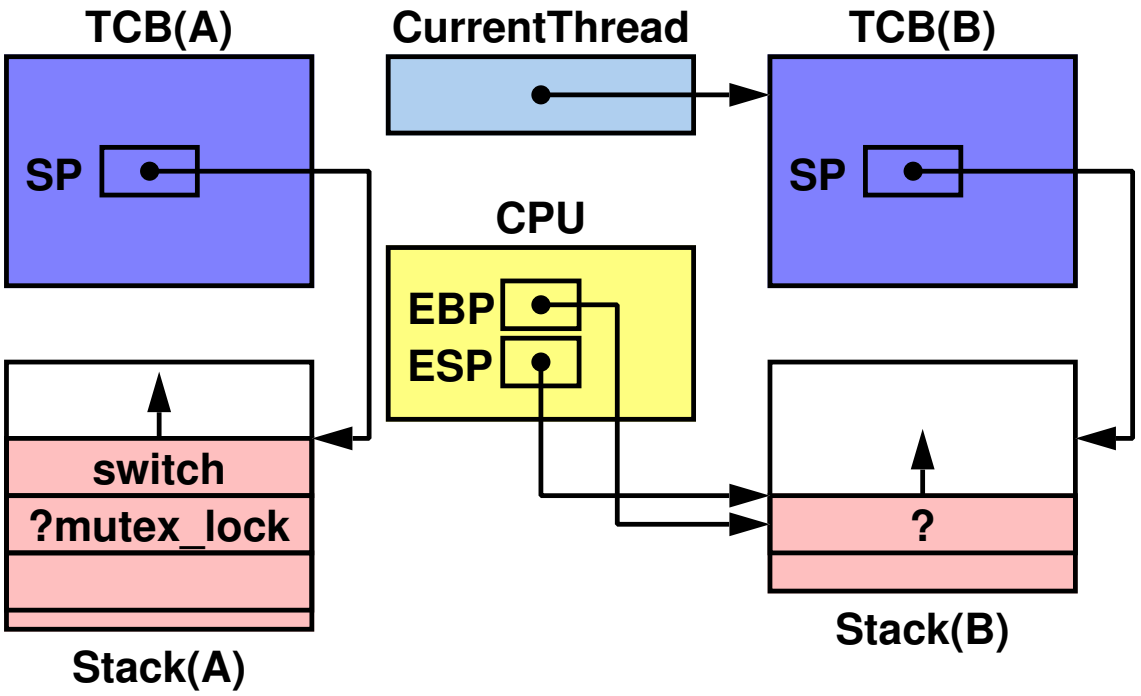


- on return from `switch()`, the registers (ebp and eip for x86) are restored into the current thread, which is the target thread!
  - which thread executes `return`?



Switching Between Threads

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



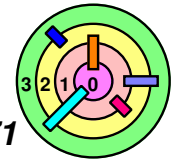
- if thread control blocks were user-space data structures, threads were switched *without* getting the kernel involved!

Note: SP field inside TCB(B) no longer tracks ESP in CPU

Switching Between Threads

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

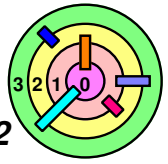
- Note: one very interesting thing happened in this call
  - usually, a single thread executes the entire procedure call
  - with `switch()`, at the beginning of the procedure call, one thread is executing
    - half way through the procedure call, another thread starts to execute
    - so, one thread 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

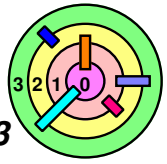
```
switch:
  ;enter switch, creating ne
  pushl %ebp ;push FP
  movl  %esp,%ebp ;set FP to
  pushl %esi  ;save esi regi
  movl  CurrentThread,%esi ;
  movl  %esp,SP(%esi) ;save
  movl  8(%ebp),CurrentThread ;store target TCB address
                              ;into CurrentThread
  movl  CurrentThread,%esi ;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;
}
```



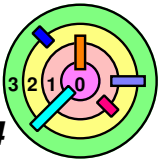
... in SPARC Assembler

```
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).
```



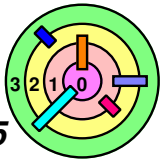
3.1 Context Switching

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



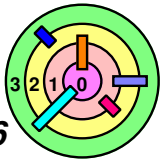
System Calls

- A system call involves the transfer of control from user code to system/kernel code and back
  - there is no thread switching!
  - depending on the OS implementation, this can view this as 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
    - in reality, more complex than just changing status
  - then it changed back to a user thread



System Calls

- Most systems provide threads with two stacks
  - one for use in user mode
  - and one for use in kernel mode
    - in some systems, one kernel stack is 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
    - the kernel cannot use the user-space stack because it cannot trust the user process

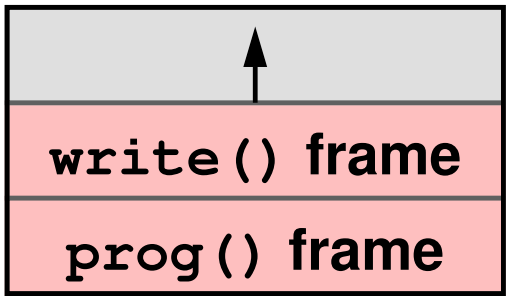


System Calls

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

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

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



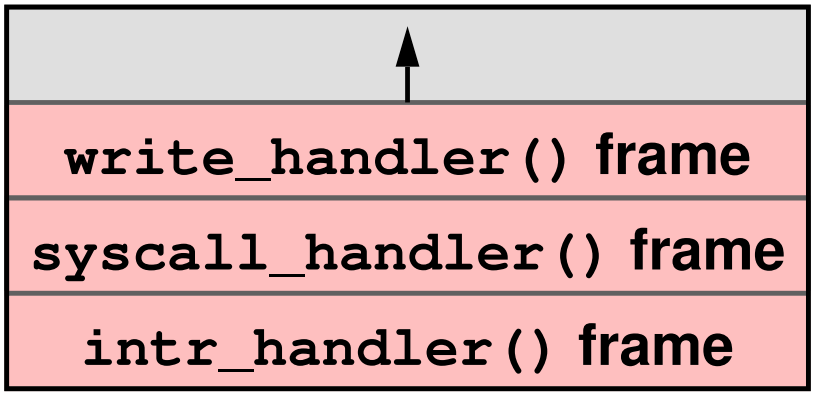
User Stack

User

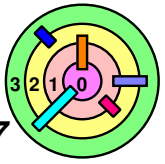
Kernel

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

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

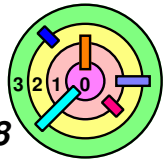


Kernel Stack



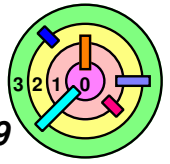
System Calls

- 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 Abstraction Layer (HAL)* save IP and SP in "temporary locations" in kernel space *(e.g., the interrupt stack)*
     - additional registers may be saved
     - *HAL* is *hardware-dependent* (outside the scope of this class)
  3) HAL sets the SP to point to the *kernel stack* designated for the corresponding user process (information from PCB)
  4) HAL sets IP to *interrupt 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*



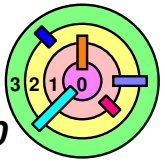
Context Switch

- The big idea here is that in order to perform a context switch, you must first save your context
  - therefore, you must know what constitutes the context
  - then you save all of it
    - what's the *minimum* amount of context to save?
    - context can be stored in several places
      - stack
      - thread control block (e.g., in a system call, the TCB contains pointers to *both* the corresponding user stack frame and the kernel stack frame)
      - etc.
  - when switching back, you must restore the context
- In general, it's difficult to make a "clean" context switch
  - when you switch from context A to context B
    - there may be time you are in the context of both A and B
    - there may be time you are in neither contexts



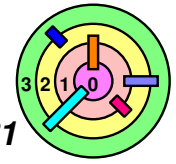
3.1 Context Switching

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



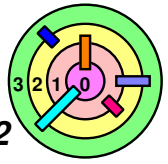
Interrupts

- Do not confuse *interrupts* with *signals* (even though the terminologies related to them are similar)
  - *signals* are *generated by the kernel*
    - they are delivered to the *user process*
    - *signal* ≠ *software interrupt*
  - *interrupts* are *generated by the hardware*
    - they are delivered to the *kernel*
      - they are delivered to the HAL and then the kernel
- When an *interrupt* occurs, the processor puts aside the current context and switch to an *interrupt context*
  - the current context can be a *thread context* or *another interrupt context*
  - when the interrupt handler is finishes, the processor generally resumes the original context



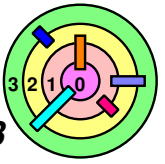
Interrupting A User Thread

- If interrupt occurs when a *user thread* is executing in the CPU
  1) *Disable interrupt* and set processor mode to *kernel mode*
  2) The *Hardware Abstraction Layer (HAL)* save IP and SP in "temporary locations" in kernel space *(e.g., the interrupt stack)*
     - additional registers may be saved
     - *HAL* is *hardware-dependent* (outside the scope of this class)
  3) HAL sets the SP to point to the *kernel stack* designated for the corresponding user process (information from PCB)
  4) HAL sets IP to *interrupt 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
- What about interrupting a *kernel thread* or an *interrupt service routine?*

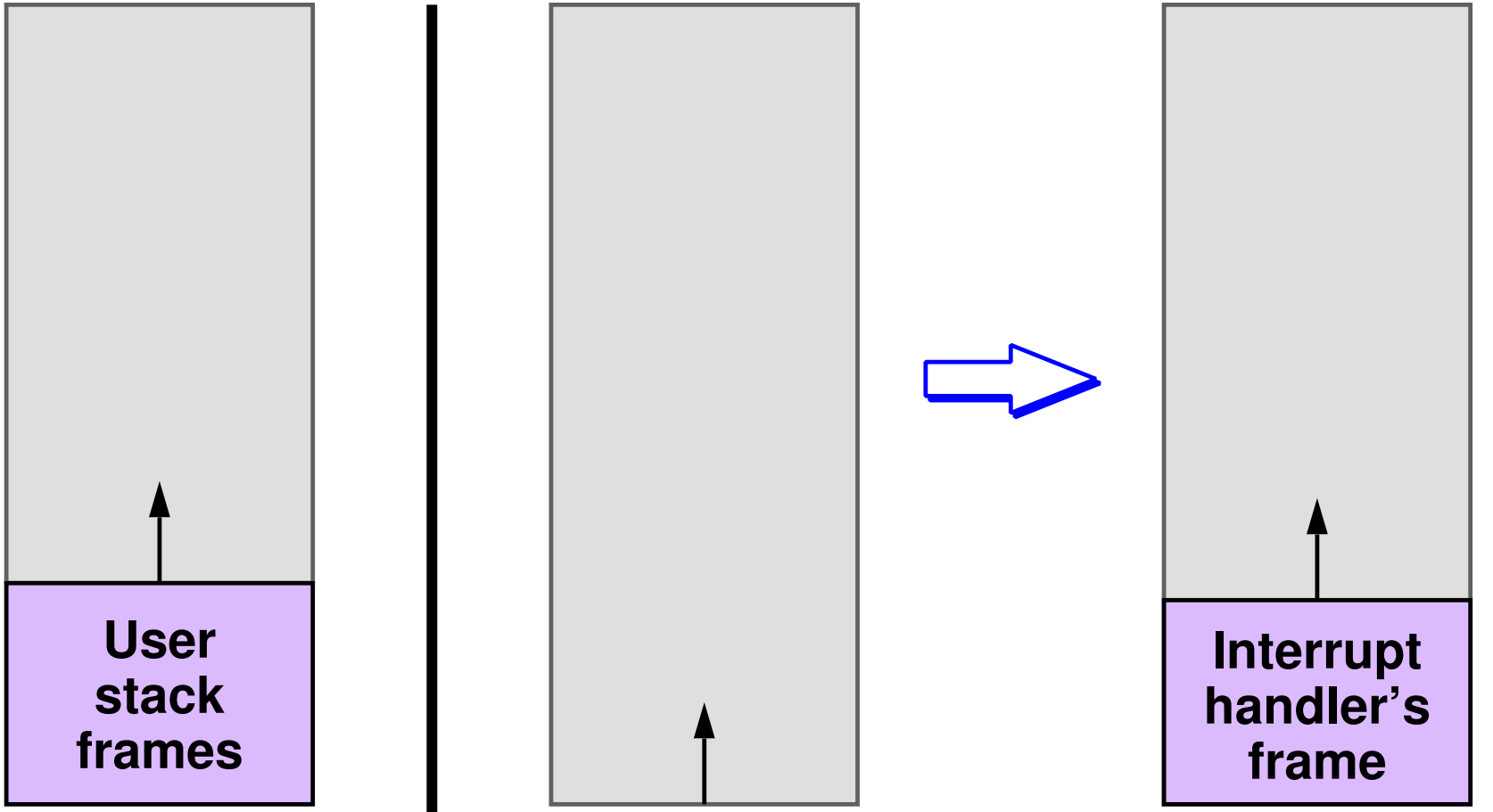


Interrupts

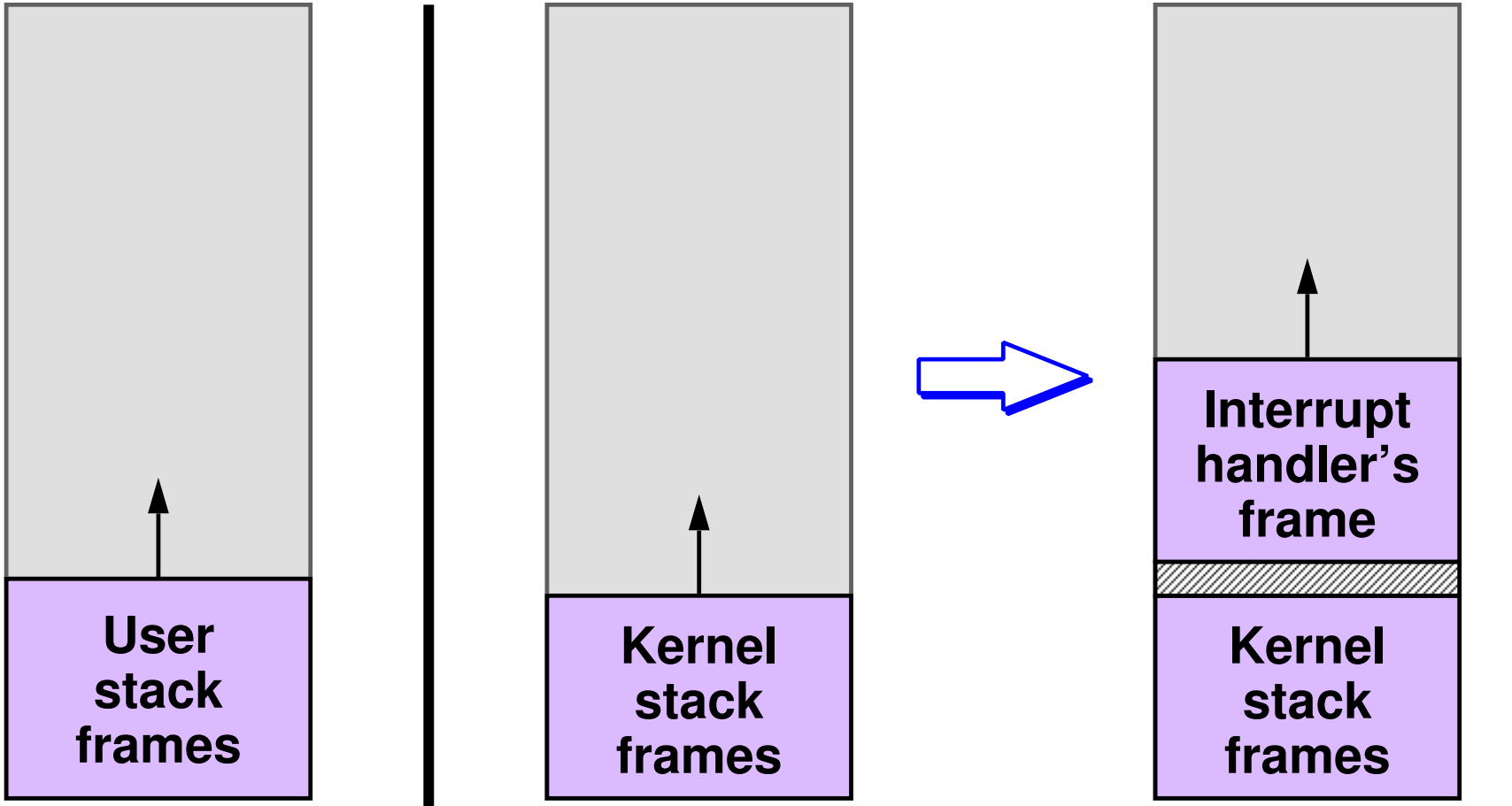
- Interrupt context needs a stack
  - which stack should it use?
  - there are several possibilities
    1) allocate a new stack each time an interrupt occurs
       - too slow
    2) have one stack shared by all interrupt handlers
       - not often done
    3) interrupt handler could borrow a stack from the thread it is interrupting
       - most common



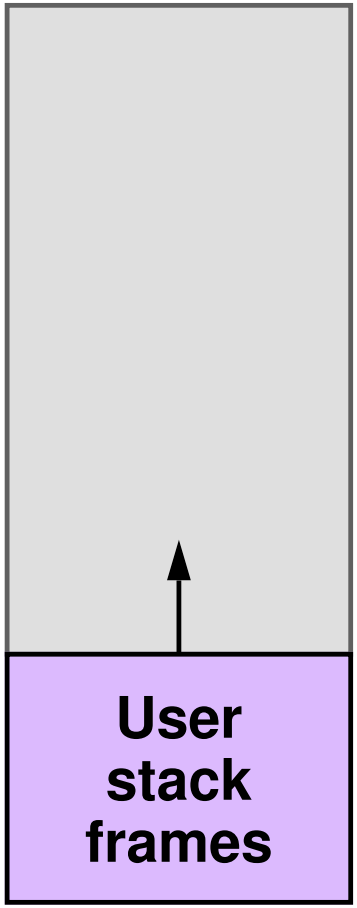
Currently Executing User Thread



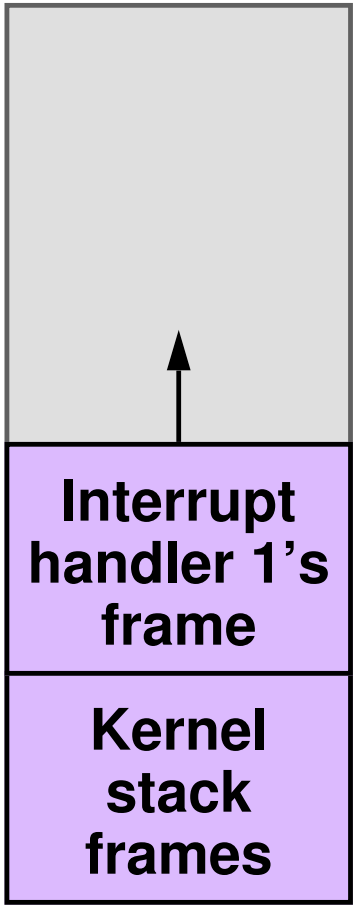
Currently Executing Kernel Thread



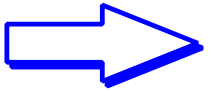
Currently Executing Another Interrupt Service Routine

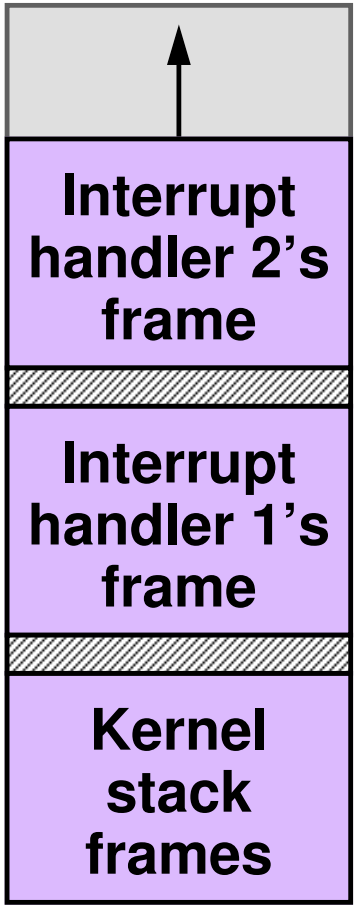


Current thread's
user stack



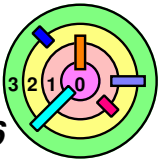
Current thread's
kernel stack





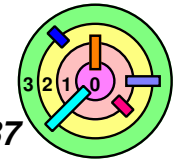
Current thread's
kernel stack

User | Kernel



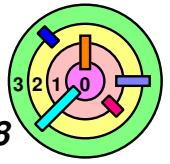
Interrupts

- For approaches (2) and (3), there is no way to suspend one interrupt handler and *resume* the execution of another
  - since there is only *one stack* for all the interrupt handlers
  - therefore, the handler of the most recent interrupt must *run to completion*
    - when it's done, the stack frame is removed, and the next-most-recent interrupt now must run to completion
  - this is a *big deal!*
    - once you have interrupt handlers running, a normal thread (no matter how important it is) cannot run until *all* interrupt handlers complete
      - this is why an interrupt service routine should do as little as possible (and figure out a way to do the rest later)
    - if we have approach (1), then we won't have this problem



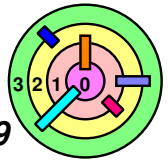
Interrupts

- What if an interrupt service routine takes too long to run?
  - interrupt handler places a description of the work that must be done on a queue of some sort, then arranges for it to be done in some other context at a later time
    - still need to do *something* in the interrupt handler
      1) *unblock a kernel thread* that's sleeping in the corresponding I/O queue
      2) *start the next I/O opertion* on the same device
  - this approach is used in many systems, including Windows and Linux
    - will discuss further in Ch 5



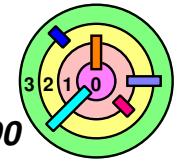
Interrupt Mask

- Interrupt can be ***masked***, i.e., temporarily blocked
  - if an interrupt occurs while it is masked, the interrupt indication remains ***pending***
  - once it is unmasked, the processor is interrupted
- How interrupts are masked is architecture-dependent
  - common approaches
    1) hardware register implements a ***bit vector / mask***
       - if a particular bit is set, the corresponding interrupt class is enable (or disabled)
       - the kernel masks interrupts by setting bits in the register
       - when an interrupt does occur, the corresponding mask bit is set in the register (block other interrupts of the same class)
       - cleared when the handler returns
    2) hierarchical interrupt levels (more common)

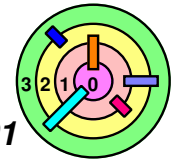


Interrupt Mask

- Interrupt can be *masked*, i.e., temporarily blocked
  - if an interrupt occurs while it is masked, the interrupt indication remains *pending*
  - once it is unmasked, the processor is interrupted
- How interrupts are masked is architecture-dependent
  - common approaches
    1) hardware register implements a *bit vector / mask*
    2) hierarchical interrupt levels (more common)
       - the processor masks interrupts by setting an *Interrupt Priority Level (IPL)* in a hardware register
       - all interrupts with the current or lower levels are masked
       - the kernel masks a class of interrupts by setting the IPL to a particular value
       - when an interrupt does occur, the current IPL is set to that of the level the interrupt belongs
       - restores to previous value on handler return

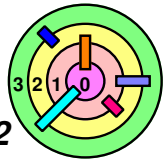


3.2 Input/Output Architectures



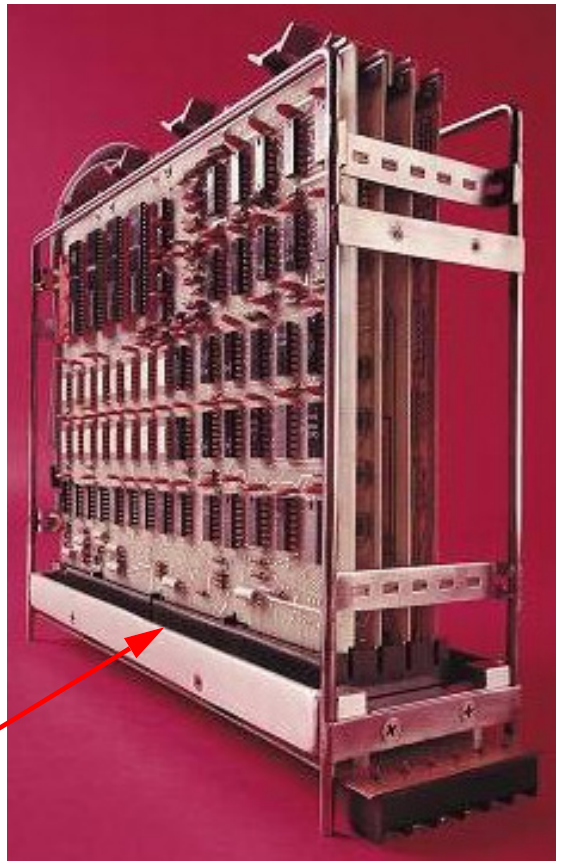
Input/Output

- Architectural concerns
  - memory-mapped I/O
    - programmed I/O (PIO)
    - direct memory access (DMA)
  - I/O processors (channels)
- Software concerns
  - device drivers
  - concurrency of I/O and computation



What Does A Computer Look Like?

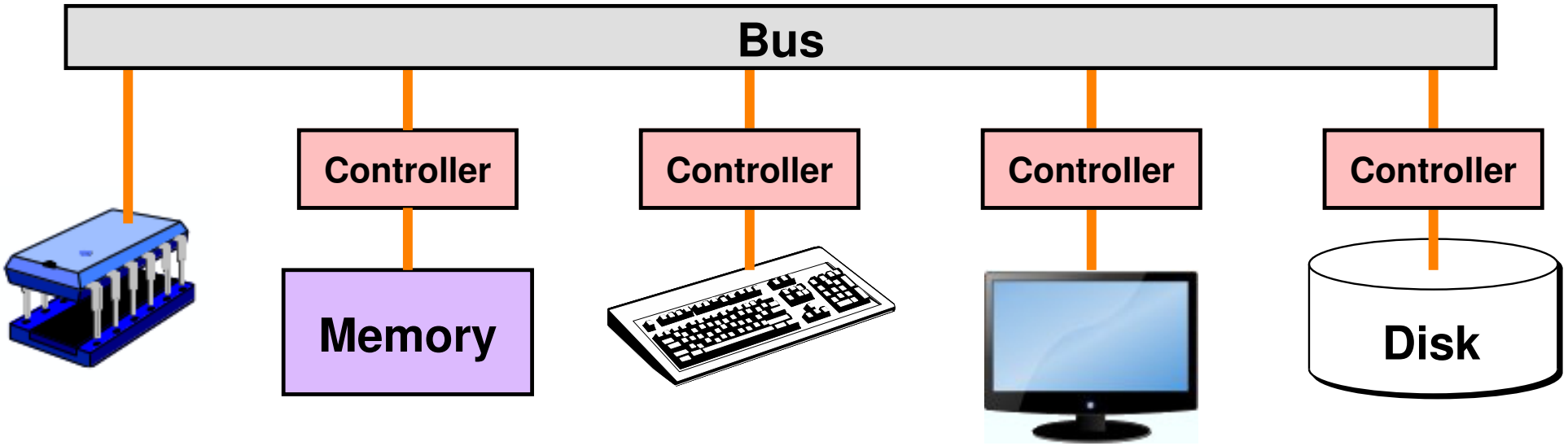
- LSI-11
  - processor for PDP-11
- Boards are connected over a "bus"
  - on the "backplane"
  - various standards for PDP-11
    - Unibus, Q-Bus, etc.



connect to backplane bus

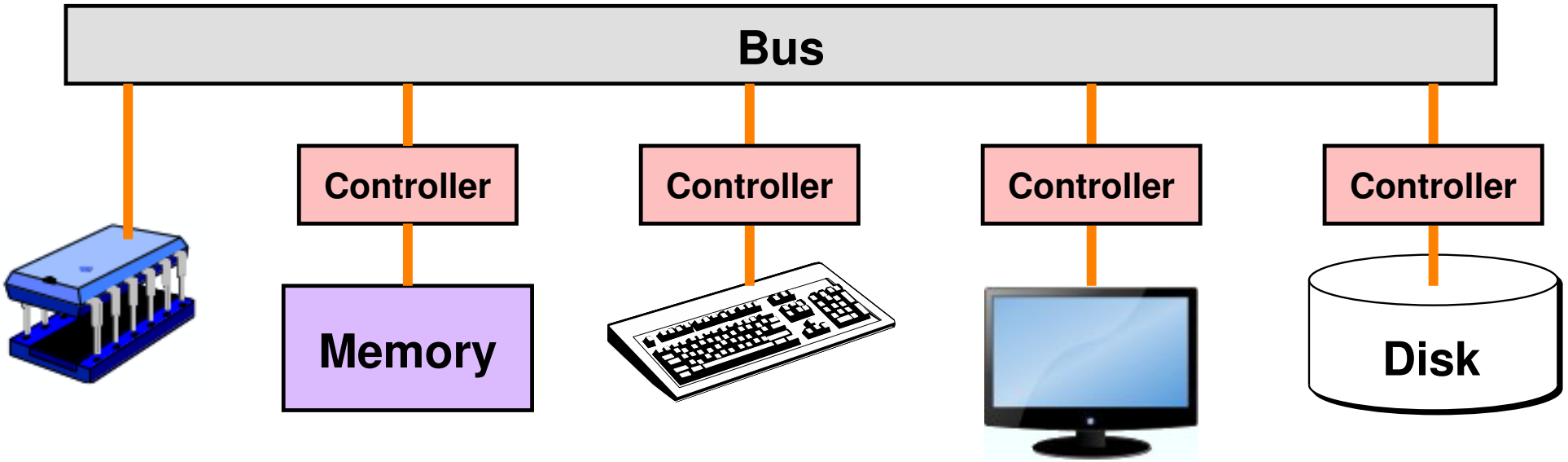
http://hampage.hu/pdp-11/lsi11.html

Simple I/O Architecture

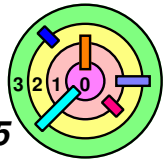


- memory-mapped I/O
  - all controllers listen on the bus to determine if a request is for itself or not
  - memory controller behaves differently from other controllers, i.e., it passes the bus request to primary memory
  - others "process" the bus request
    - and respond to relatively few addresses
  - memory is not really a "device"

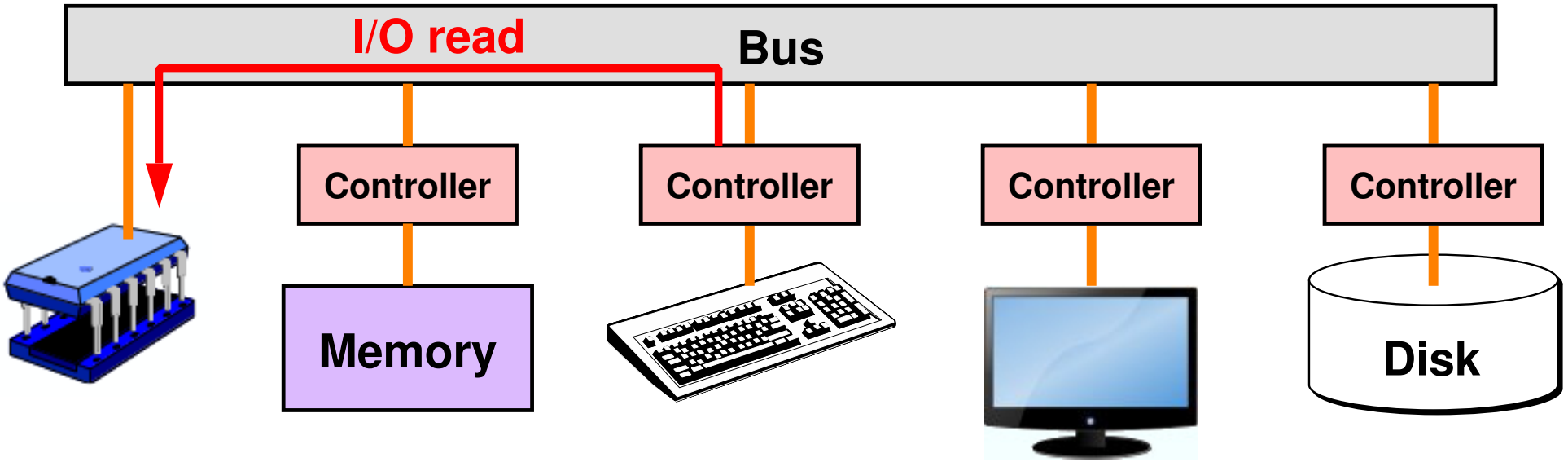
Simple I/O Architecture



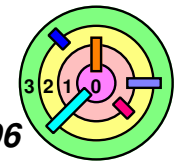
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus



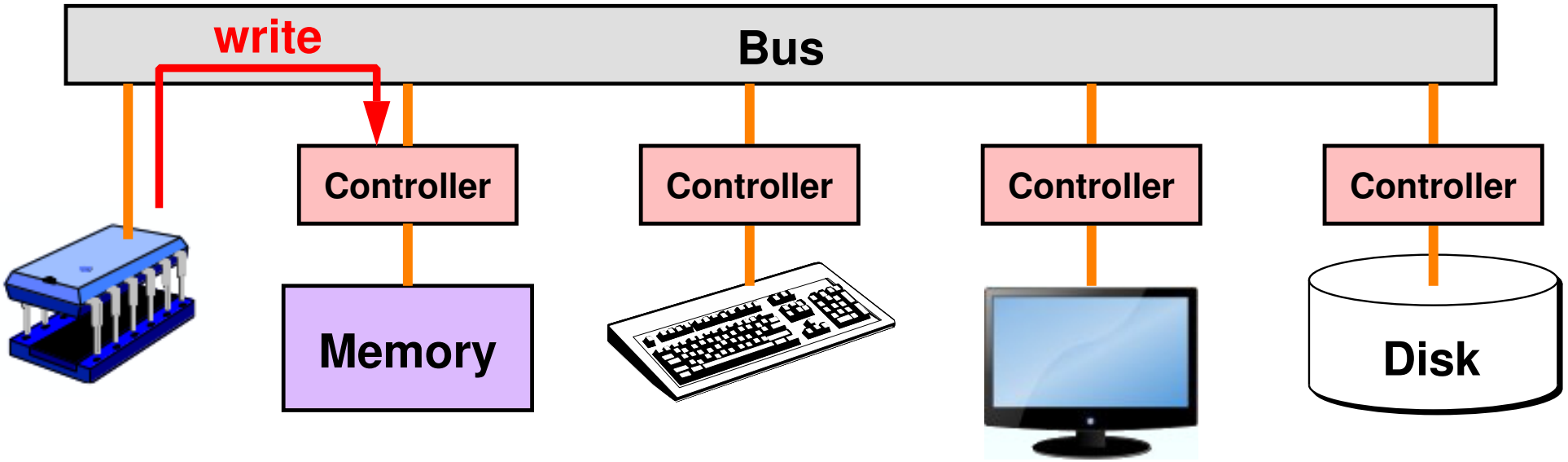
Simple I/O Architecture



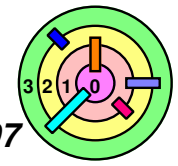
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus



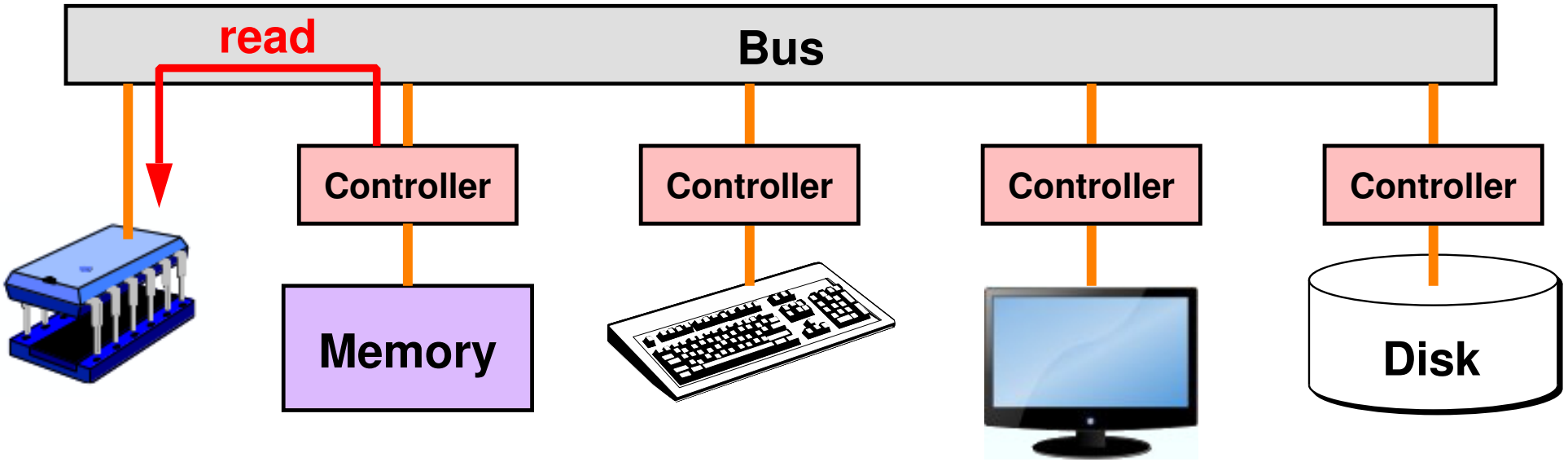
Simple I/O Architecture



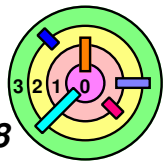
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus



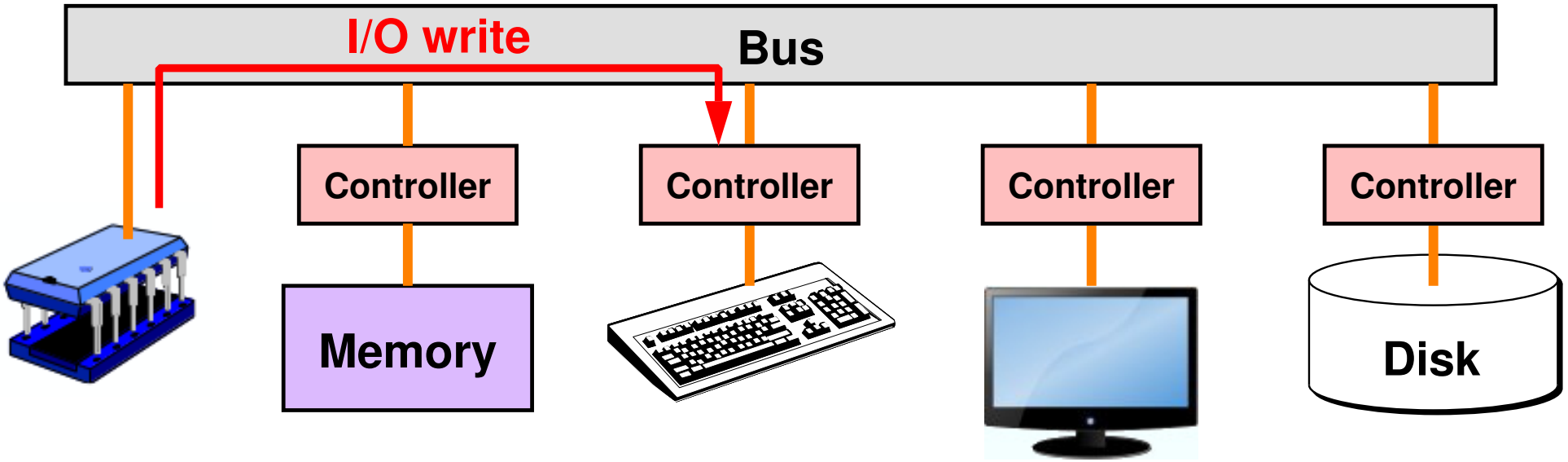
Simple I/O Architecture



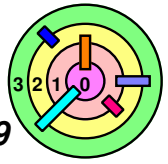
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus



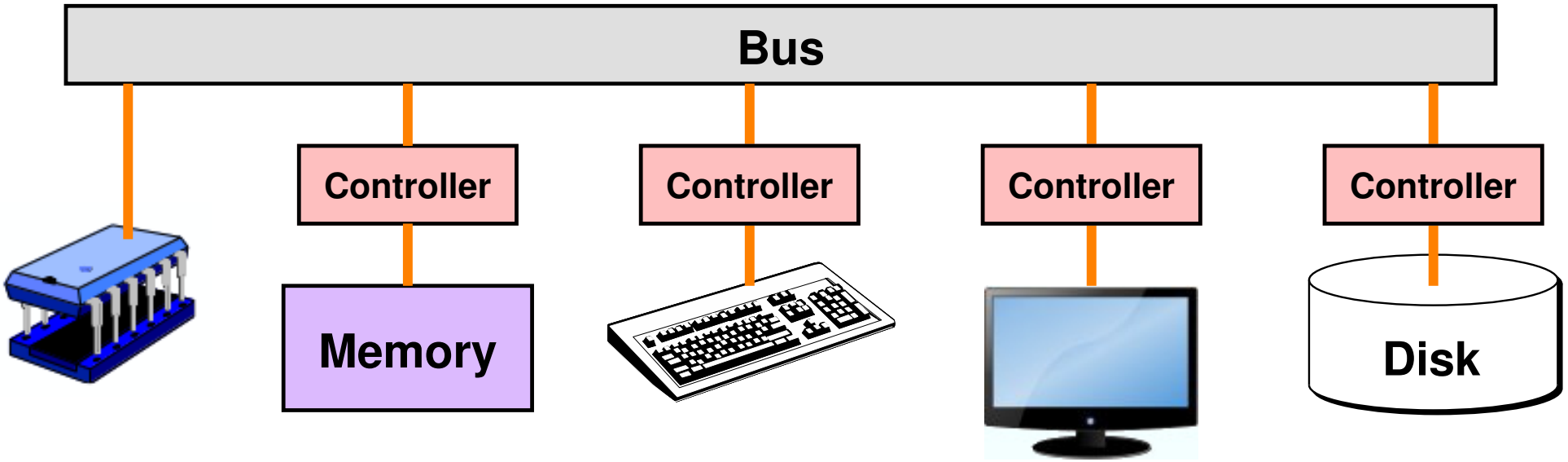
Simple I/O Architecture



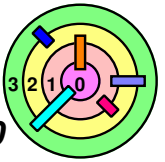
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus



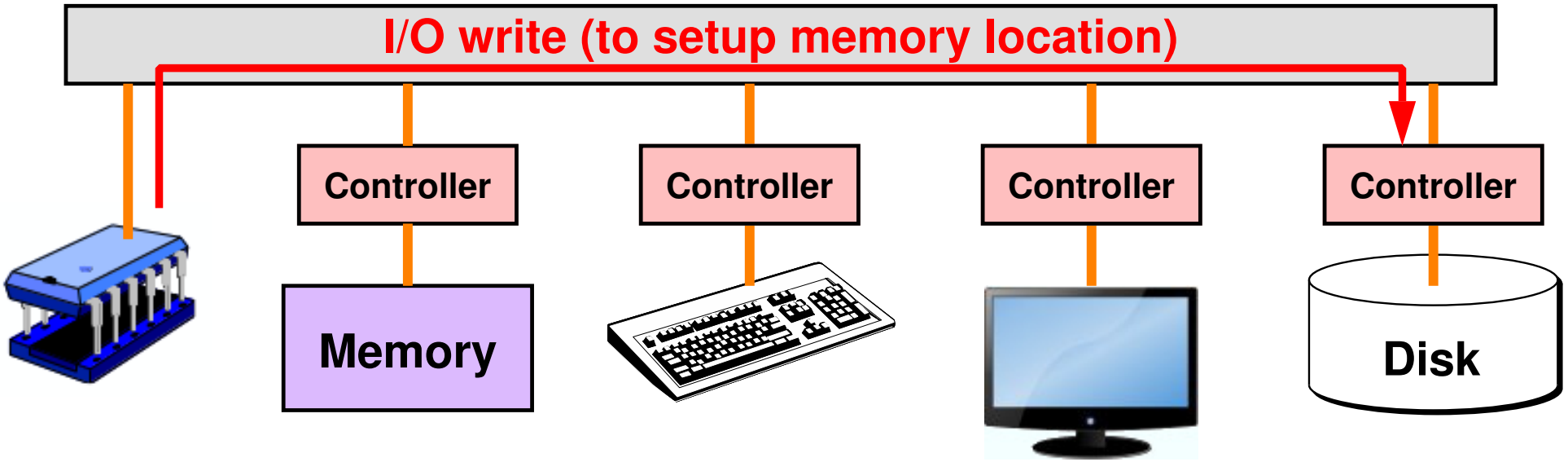
Simple I/O Architecture



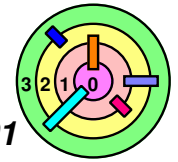
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
  - DMA (direct memory access)
    - the controller performs the I/O itself
    - the processor writes to the controller to tell it where to transfer the results to
    - the controller takes over and transfers data between itself and primary memory



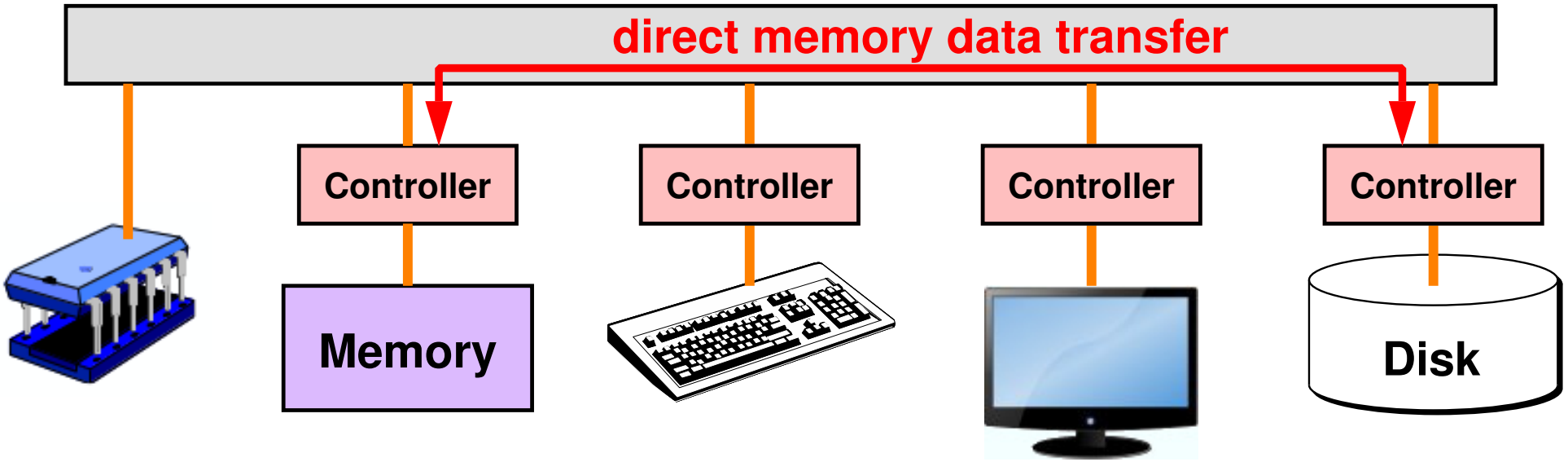
Simple I/O Architecture



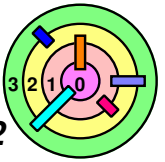
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
  - DMA (direct memory access)
    - the controller performs the I/O itself
    - the processor writes to the controller to tell it where to transfer the results to
    - the controller takes over and transfers data between itself and primary memory



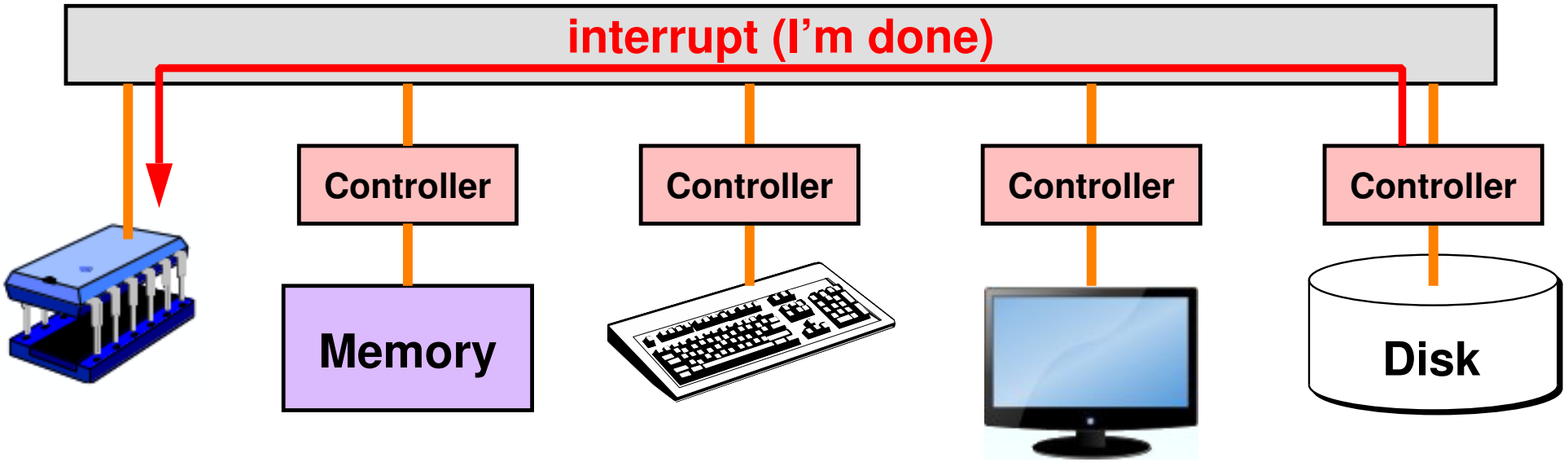
Simple I/O Architecture



- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
  - DMA (direct memory access)
    - the controller performs the I/O itself
    - the processor writes to the controller to tell it where to transfer the results to
    - the controller takes over and transfers data between itself and primary memory



Simple I/O Architecture



- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
  - DMA (direct memory access)
    - the controller performs the I/O itself
    - the processor writes to the controller to tell it where to transfer the results to
    - the controller takes over and transfers data between itself and primary memory

PIO Registers

- This is the abstraction of a PIO device
  - a *"register"* is just a *memory-mapped I/O address* on the bus

GoR	GoW	IER	IEW				
RdyR	RdyW						

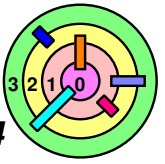
Control register (1 byte)

Status register (1 byte)

Read register (1 byte)

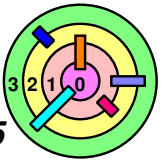
Write register (1 byte)

Legend:	GoR	Go read (start a read operation)
	GoW	Go write (start a write operation)
	IER	Enable read-completion interrupts
	IEW	Enable write-completion interrupts
	RdyR	Ready to read
	RdyW	Ready to write



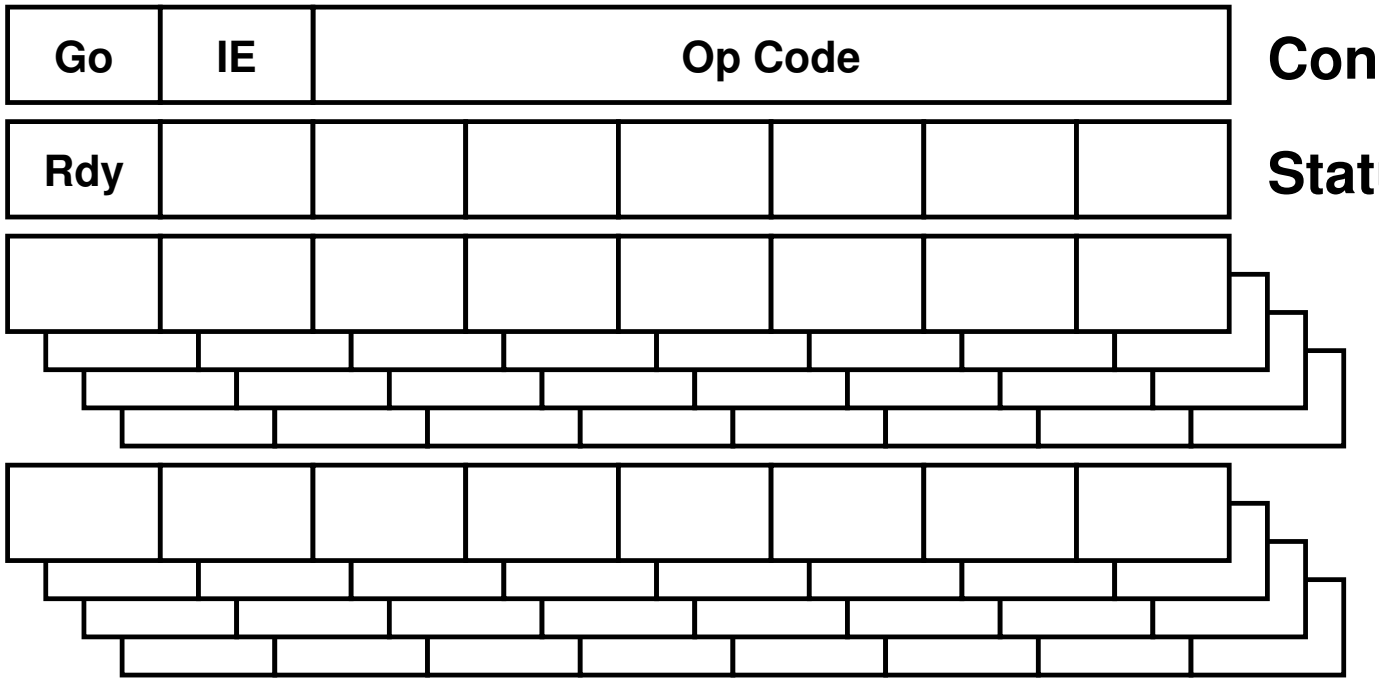
Programmed I/O

- E.g.: Terminal controller
- Procedure (write)
  - write a byte into the write register
  - set the GoW bit (and optionally the IEW bit if you'd like to be notified via an interrupt) in the control register
  - poll and wait for RdyW bit (in status register) to be set (if interrupts have been enabled, an interrupt occurs when this happens)



DMA Registers

- This is the abstraction of a DMA device
  - a *"register"* is just a *memory-mapped I/O address* on the bus



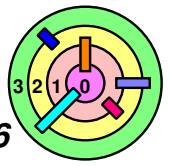
Control register (1 byte)

Status register (1 byte)

Memory address register (4 bytes)

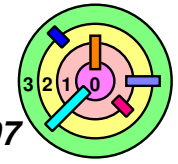
Device address register (4 bytes)

Legend:	Go	Start an operation
	Op Code	Operation code (identifies the operation)
	IE	Enable interrupts
	Rdy	Controller is ready

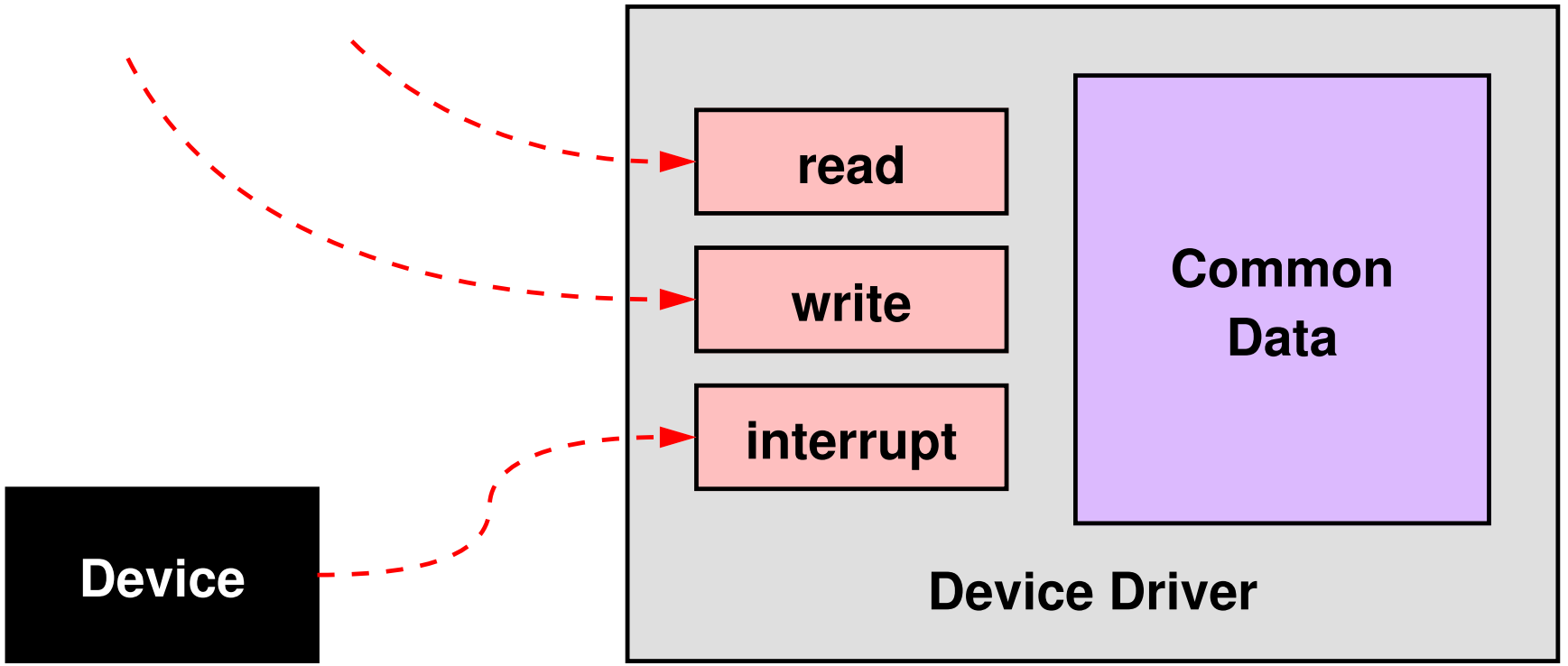


Direct Memory Access

- E.g.: Disk controller
- Procedure
  - set the disk address in the device address register (only relevant for a seek request)
  - set the buffer address in the memory address register
  - set the op code (SEEK, READ or WRITE), the Go bit and, if desired, the IE bit in the control register
  - wait for interrupt or for Rdy bit to be set



Device Drivers

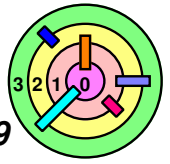


- device drivers provide a standard interface to the rest of the OS
  - *code in device drivers* knows how to talk to devices (the rest of the OS really doesn't know how to talk to devices)
  - OS can treat I/O in a *device-indepdendent* manner using an *array of function pointers*

... in C++

```
class disk {
  public:
    virtual status_t read(request_t) = 0;
    virtual status_t write(request_t) = 0;
    virtual status_t interrupt( ) = 0;
};
```

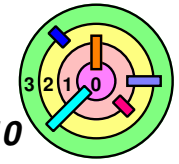
- C++ *polymorphism* achieved using virtual base class
  - each type of disk driver is a *subclass* of the disk class and has its own implementation of these functions
    - each disk driver looks like a generic disk to the OS
  - this gets compiled into an array of function pointers (which is what C++ code gets compiled into)
    - in reality, there are no object classes and no polymorphism
      - the CPU doesn't even know about data structures
      - the CPU only knows about memory addresses and how to execute machine instructions



... in C++

```
class disk {
  public:
    virtual status_t read(request_t) = 0;
    virtual status_t write(request_t) = 0;
    virtual status_t interrupt( ) = 0;
};
```

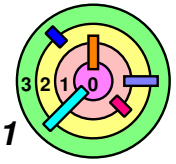
- This is a synchronous interface
  - a user thread would call the `read/write()` method
  - this starts the device and the user thread would block
  - the device driver's interrupt method is called in the interrupt context
    - if I/O is completed, the thread is unblocked and return from the `read/write()` method



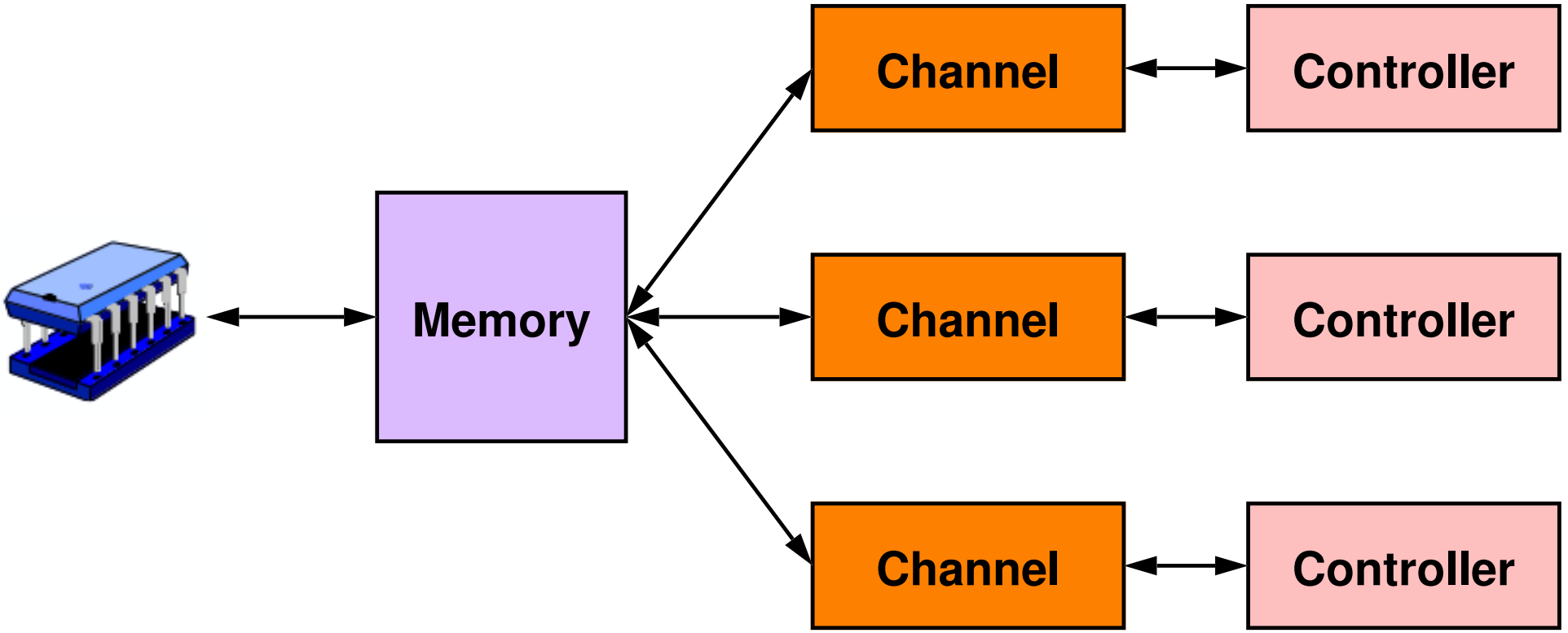
A Bit More Realistic

```
class disk {
  public:
    virtual handle_t start_read(request_t) = 0;
    virtual handle_t start_write(request_t) = 0;
    virtual status_t wait(handle_t) = 0;
    virtual status_t interrupt( ) = 0;
};
```

- Even in Sixth-Edition Unix, the internal driver interface is often asynchronous
  - `start_read/start_write()` returns a handle identifying the operation that has started
  - a thread can call the `wait()` method to synchronously wait for I/O completion
    - it's possible for multiple threads to invoke `wait()` with the same handle, if they all want the same block from a file



I/O Processors: Channels



- when I/O costs dominate computation costs
  - use I/O processors (a.k.a. channels) to handle much of the I/O work
  - important in large data-processing applications
- can even download program into a channel