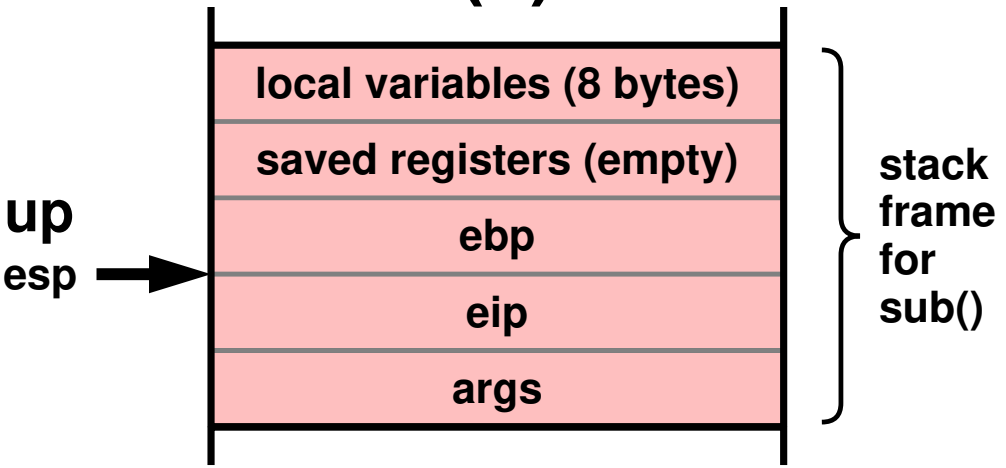
Intel x86: Subroutine Code (2)

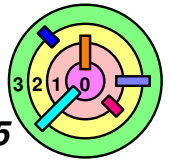
```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)
   movl $0, -8(%ebp)
   movl -4(%ebp), %ecx
   movl -8(%ebp), %eax
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp),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

set up

restore

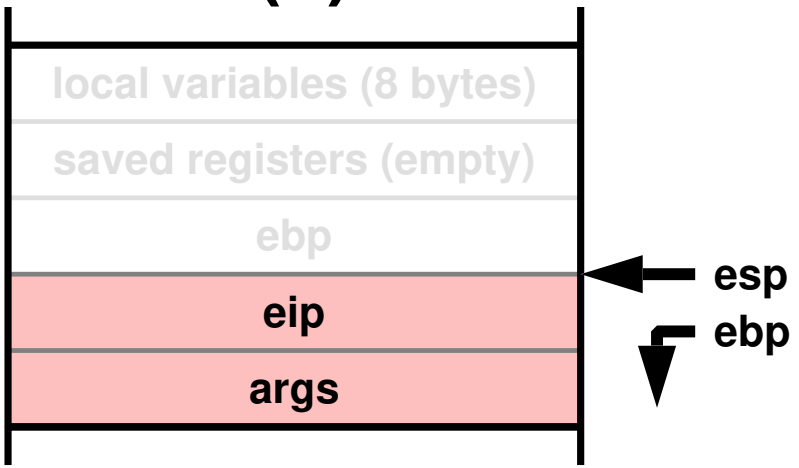



```
int sub(int x, int y) {
   // computes x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

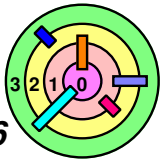


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp                  } enter
   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             } leave
   popl %ebp
   ret
```




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

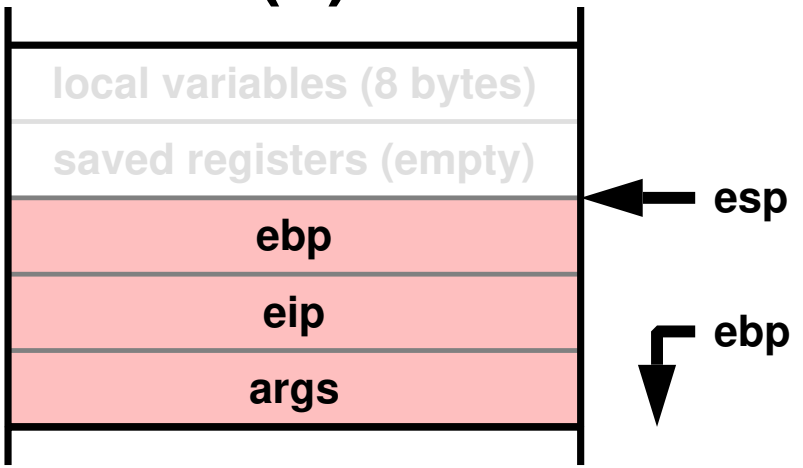


Intel x86: Subroutine Code (2)

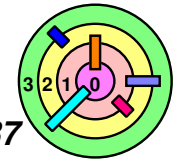
```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)
   movl $0, -8(%ebp)
   movl -4(%ebp), %ecx
   movl -8(%ebp), %eax
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp),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

set up

restore

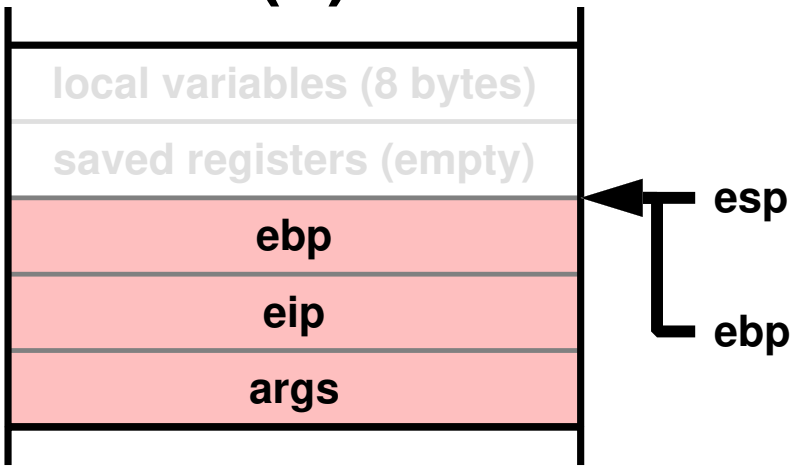



```
int sub(int x, int y) {
   // computes x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

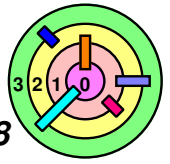


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp                  }
   movl %esp, %ebp             }  set up
   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             }  restore
   popl %ebp                   }
   ret
```

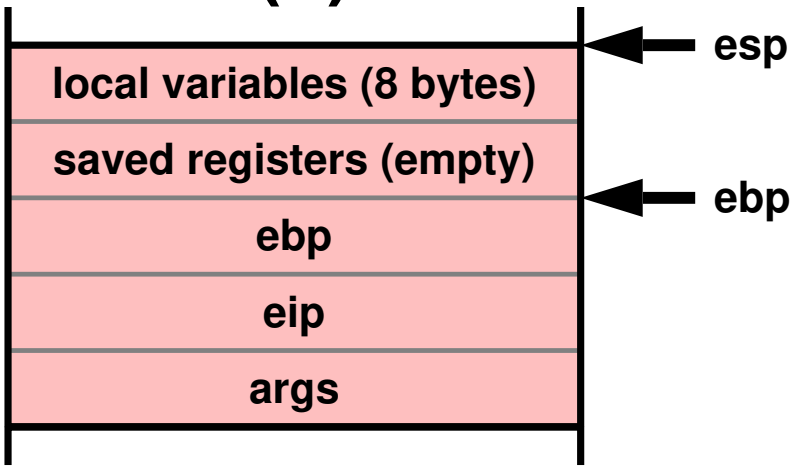



```
int sub(int x, int y) {
   // computes x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

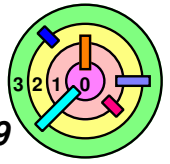


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp                    } set up
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)             } not local var
   movl $0, -8(%ebp)               initialization
   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           } restore
   movl %ebp, %esp
   popl %ebp
   ret
```

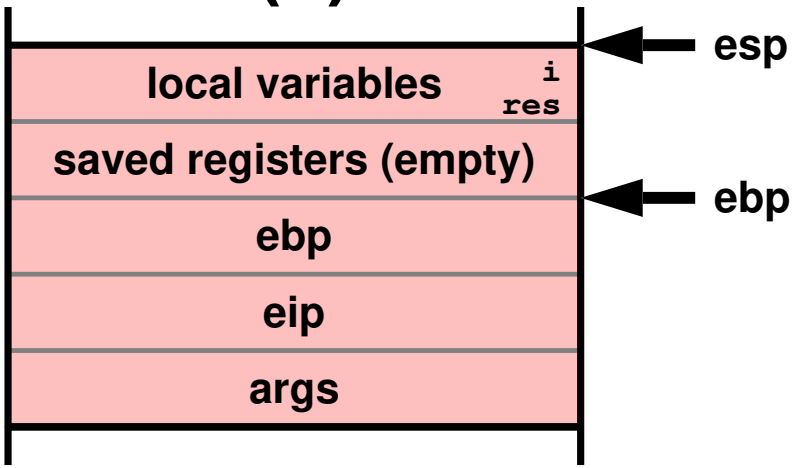



```
int sub(int x, int y) {
  // computes x^y
  int i;
  int result = 1;
  for (i=0; i<y; i++)
    result *= x;
  return(result);
}
```

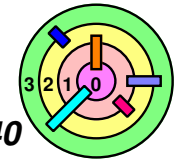


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp                    } set up
   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           } restore
   movl %ebp, %esp
   popl %ebp
   ret
```

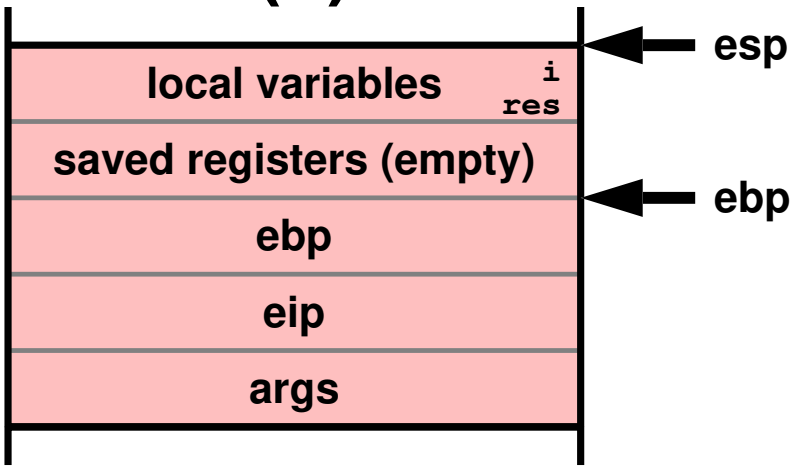



```
int sub(int x, int y) {
  // computes x^y
  int i;
  int result = 1;
  for (i=0; i<y; i++)
    result *= x;
  return(result);
}
```

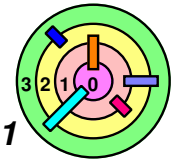


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp                    }
   movl %esp, %ebp               }  set up
   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               }  restore
   popl %ebp                     }
   ret
```

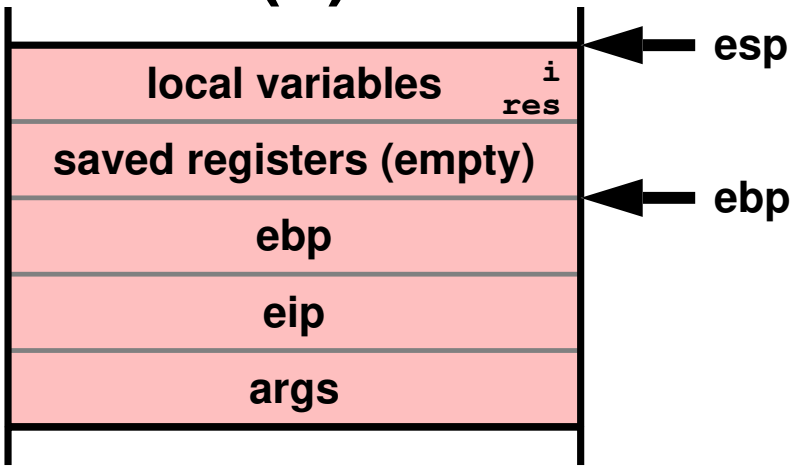



```
int sub(int x, int y) {
   // computes x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

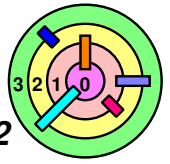


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp                    } set up
   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           } restore
   movl %ebp, %esp
   popl %ebp
   ret
```

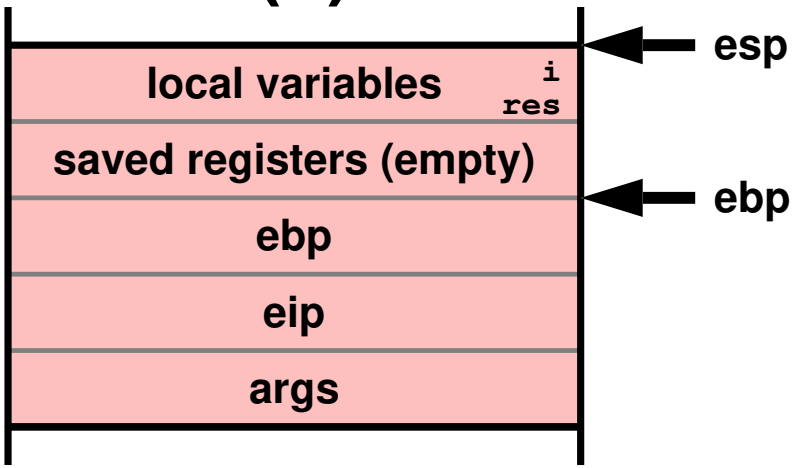



```
int sub(int x, int y) {
  // computes x^y
  int i;
  int result = 1;
  for (i=0; i<y; i++)
    result *= x;
  return(result);
}
```

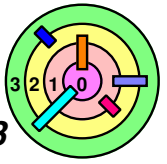


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp                    } set up
   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           } restore
   movl %ebp, %esp
   popl %ebp
   ret
```

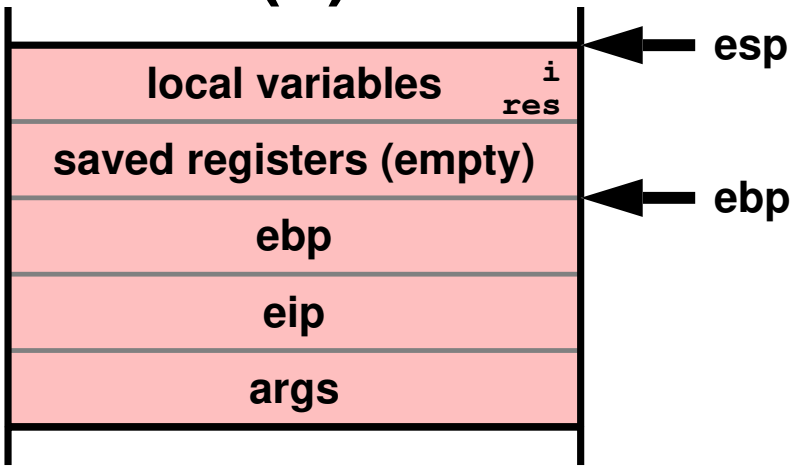



```
int sub(int x, int y) {
   // computes x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

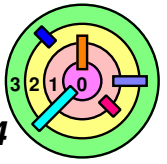


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp                    } set up
   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           } restore
   movl %ebp, %esp
   popl %ebp
   ret
```

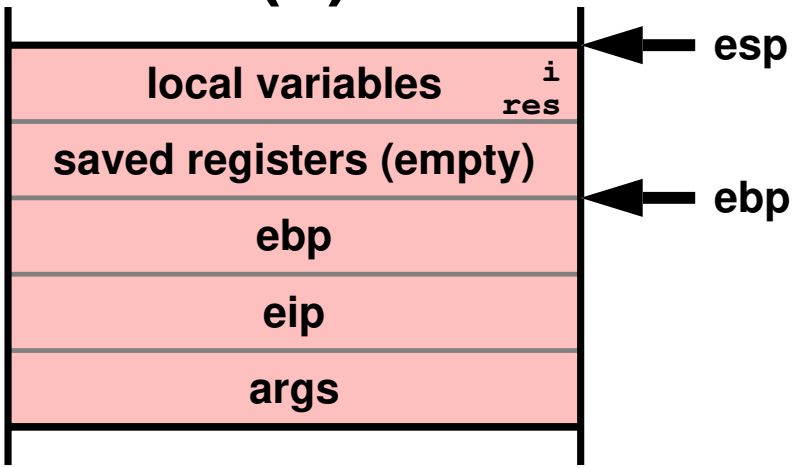



```
int sub(int x, int y) {
   // computes x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

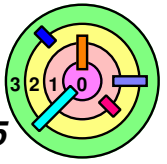


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp                    } set up
   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           } restore
   movl %ebp, %esp               }
   popl %ebp                     }
   ret
```

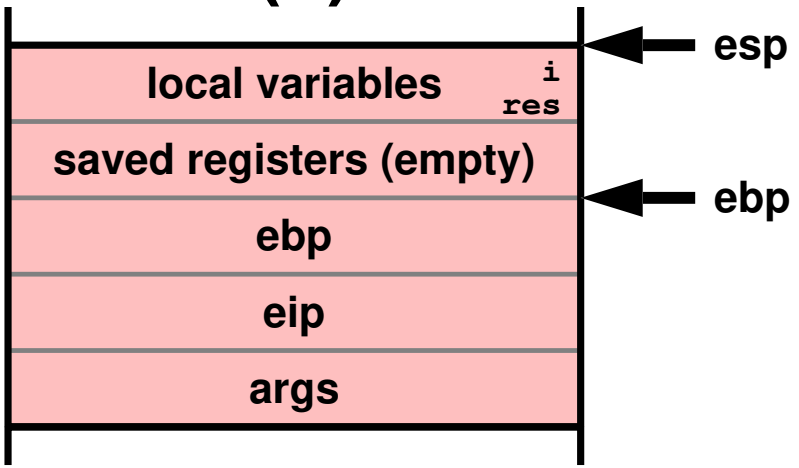



```
int sub(int x, int y) {
   // computes x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

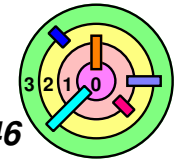


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp                    } set up
   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           } restore
   movl %ebp, %esp               }
   popl %ebp                     }
   ret
```

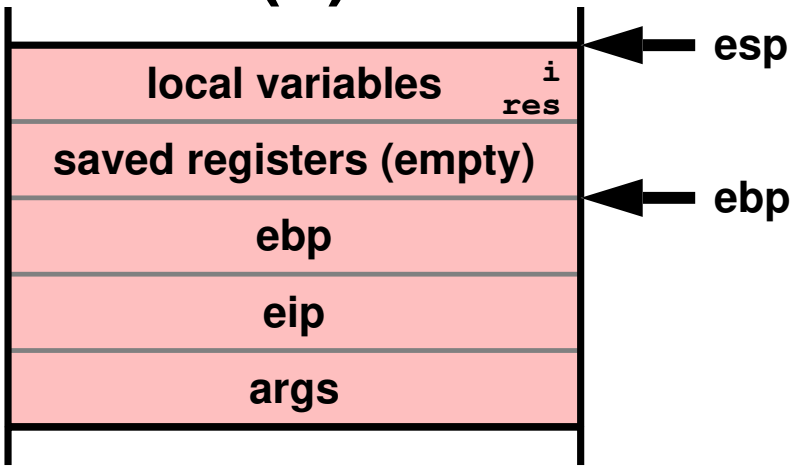



```
int sub(int x, int y) {
   // computes x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

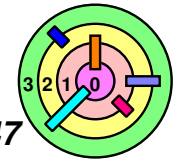


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp                    } set up
   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           } restore
   movl %ebp, %esp
   popl %ebp
   ret
```

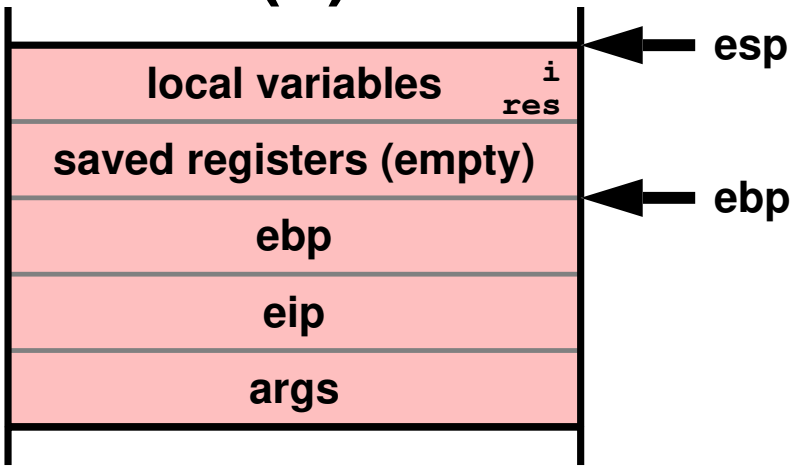



```
int sub(int x, int y) {
   // computes x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

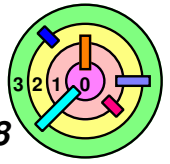


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp                    }
   movl %esp, %ebp               }  set up
   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               }  restore
   popl %ebp                     }
   ret
```

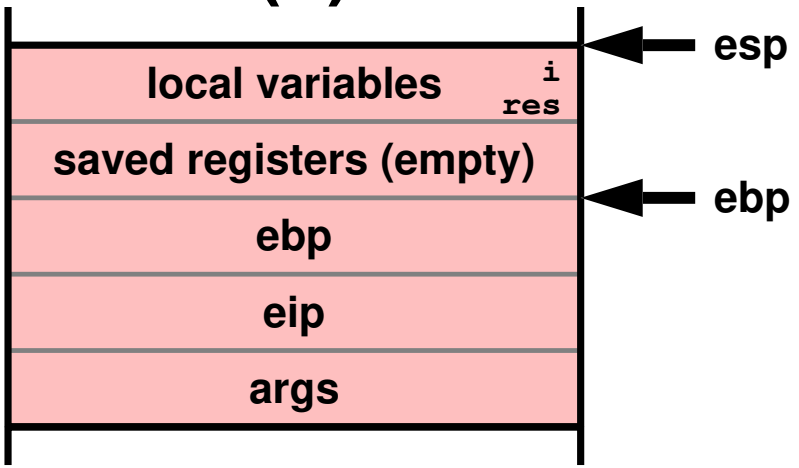



```
int sub(int x, int y) {
   // computes x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

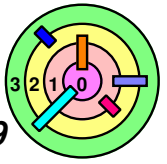


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp                    } set up
   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           } restore
   movl %ebp, %esp
   popl %ebp
   ret
```

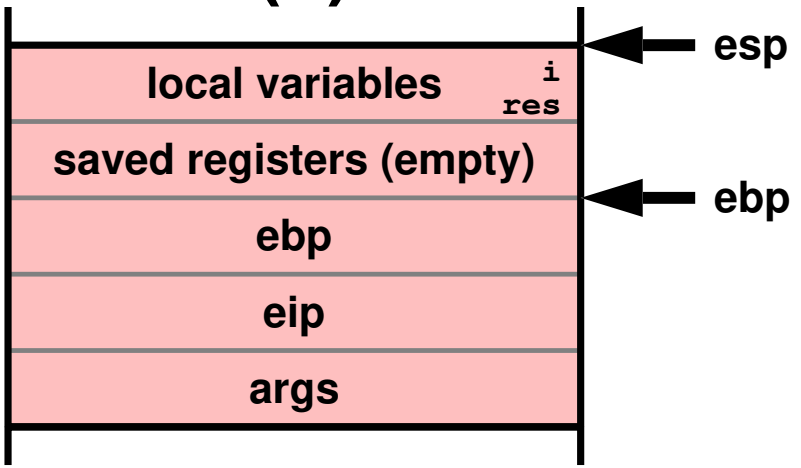



```
int sub(int x, int y) {
   // computes x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

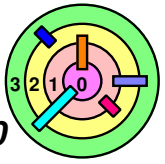


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp                    } set up
   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           } restore
   movl %ebp, %esp               }
   popl %ebp                     }
   ret
```




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

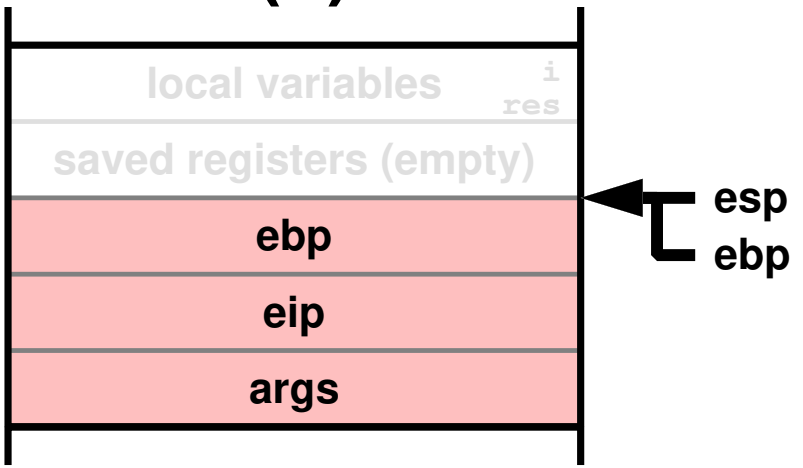


Intel x86: Subroutine Code (2)

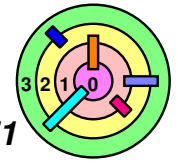
```
sub:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   movl $1, -4(%ebp)
   movl $0, -8(%ebp)
   movl -4(%ebp), %ecx
   movl -8(%ebp), %eax
beginloop:
   cmpl 12(%ebp), %eax
   jge endloop
   imull 8(%ebp), %ecx
   addl $1, %eax
   jmp beginloop
endloop:
   movl %ecx, -4(%ebp)
   movl -4(%ebp),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

set up

restore

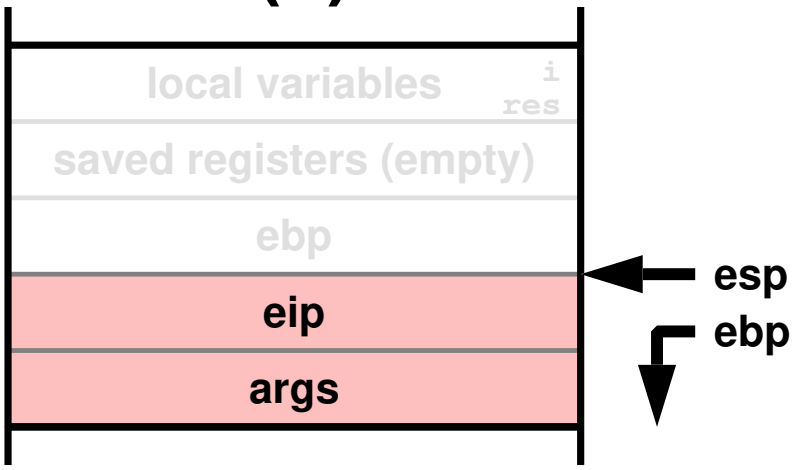



```
int sub(int x, int y) {
   // computes x^y
   int i;
   int result = 1;
   for (i=0; i<y; i++)
      result *= x;
   return(result);
}
```

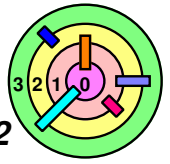


Intel x86: Subroutine Code (2)

```
sub:
   pushl %ebp                    } set up
   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           } restore
   movl %ebp, %esp               }
   popl %ebp                     }
   ret
```




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



Intel x86: Subroutine Code (1)

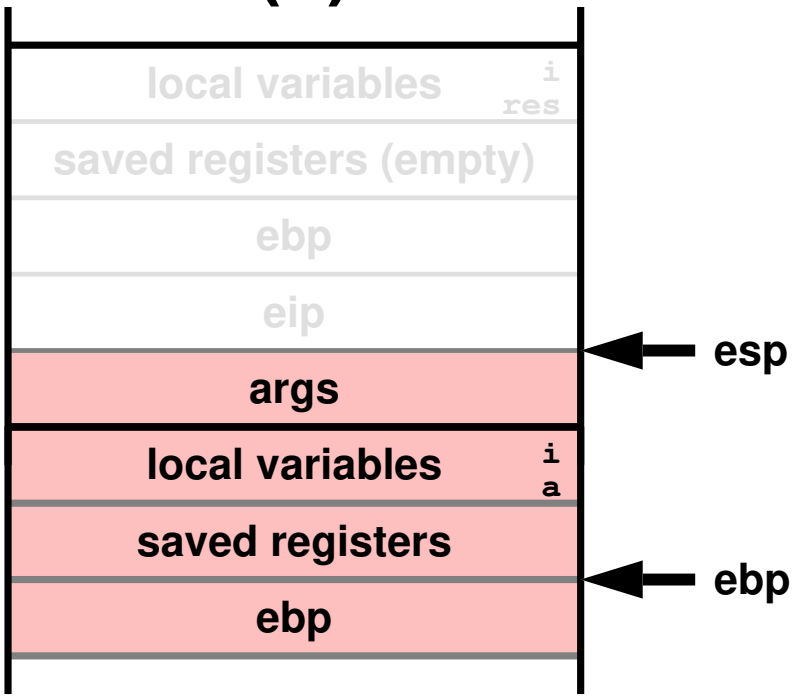
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret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

set up stack frame

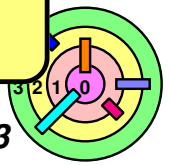
push args

pop args; get result

set return value and restore frame




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



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

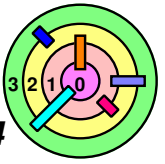
Output Registers

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

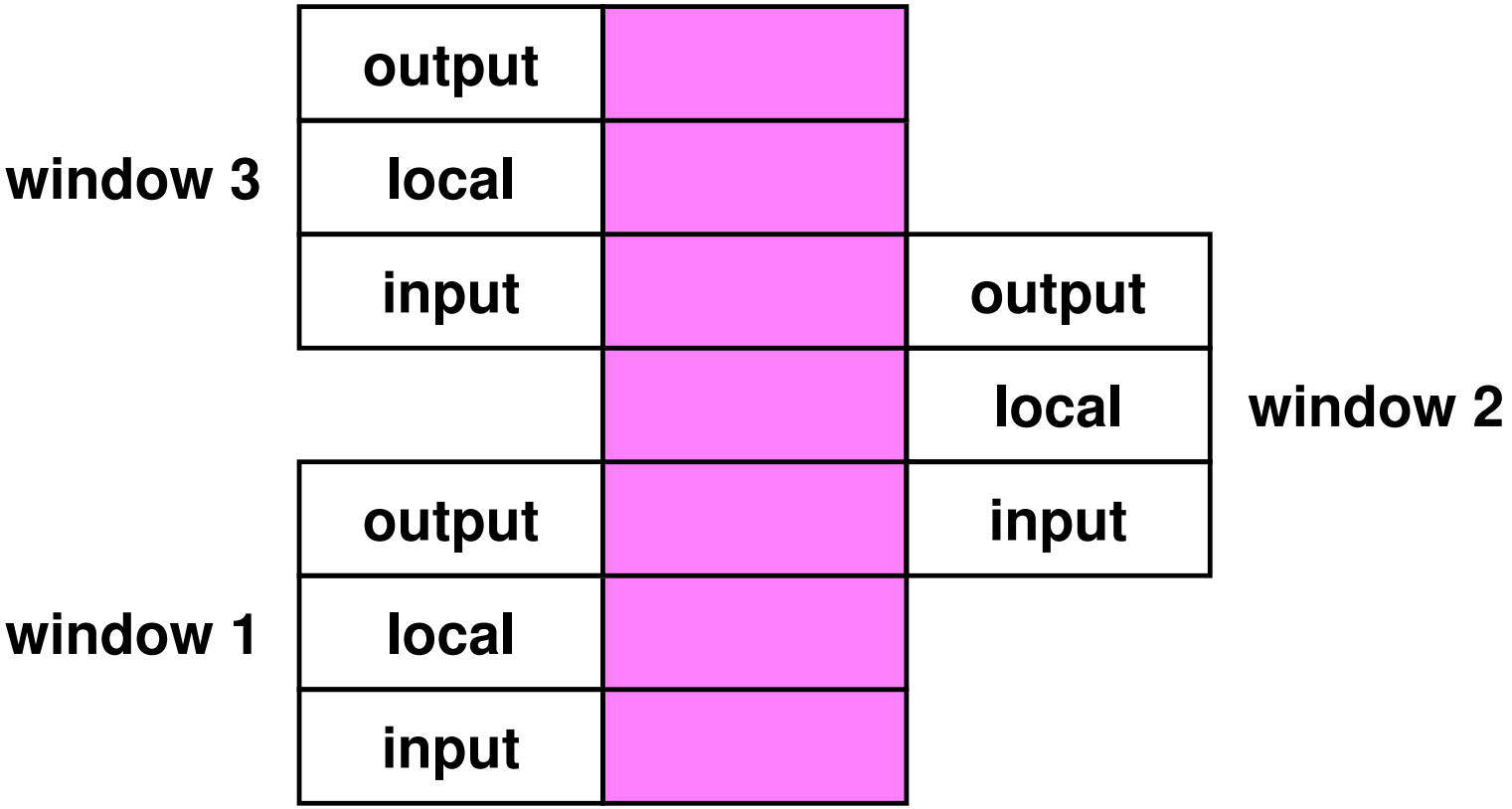
Local Registers

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

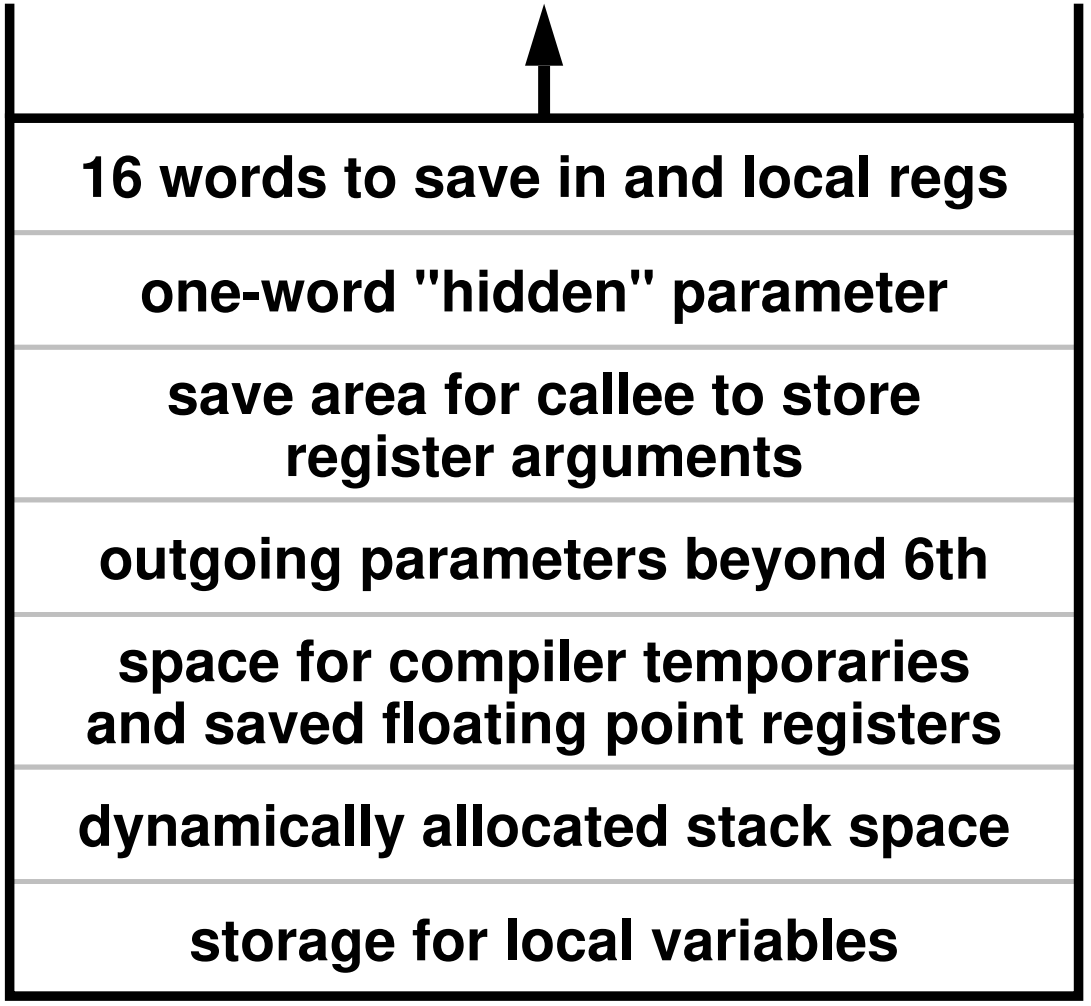
Global Registers



SPARC Architecture: Register Windows

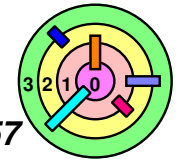


SPARC Architecture: Stack



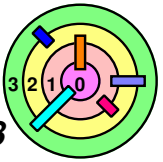
SPARC Architecture: Subroutine Code

```
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)
```



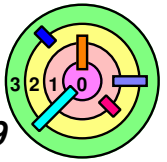
3.1 Context Switching

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



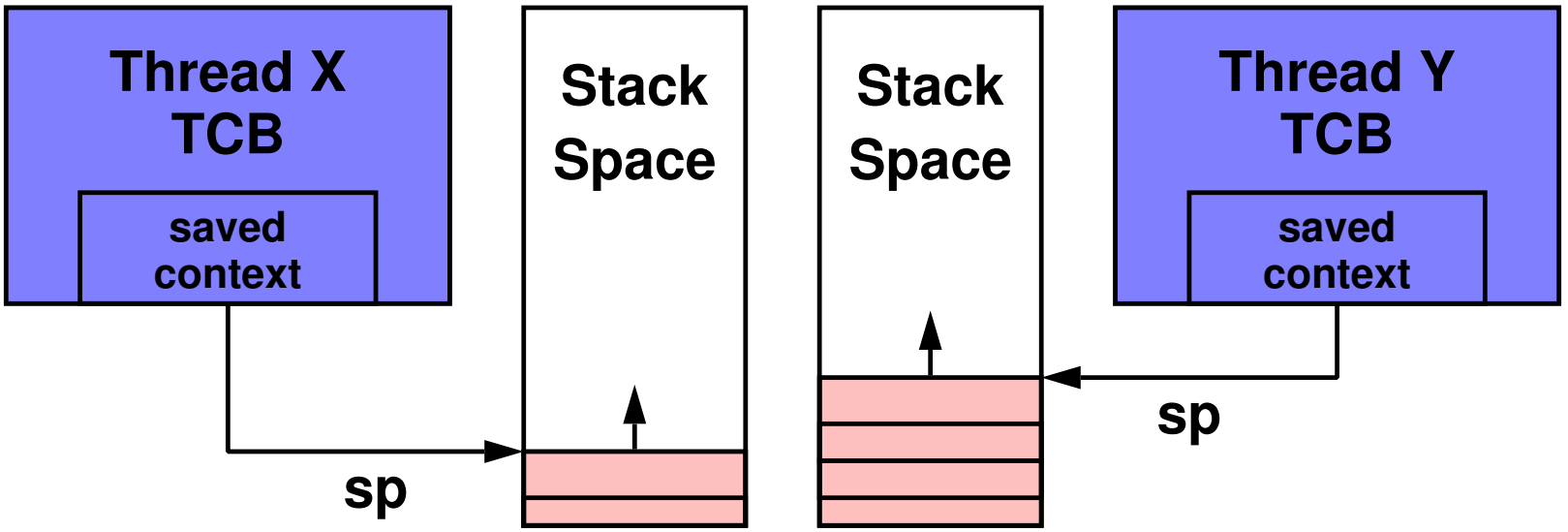
Threads & Coroutines

- 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*

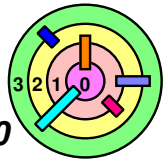


Representing Threads

- A thread's context
  - its *stack*, its *register state*
    - can be stored in a *Thread Control Block (TCB)*
    - this is what *sleeping* threads look like

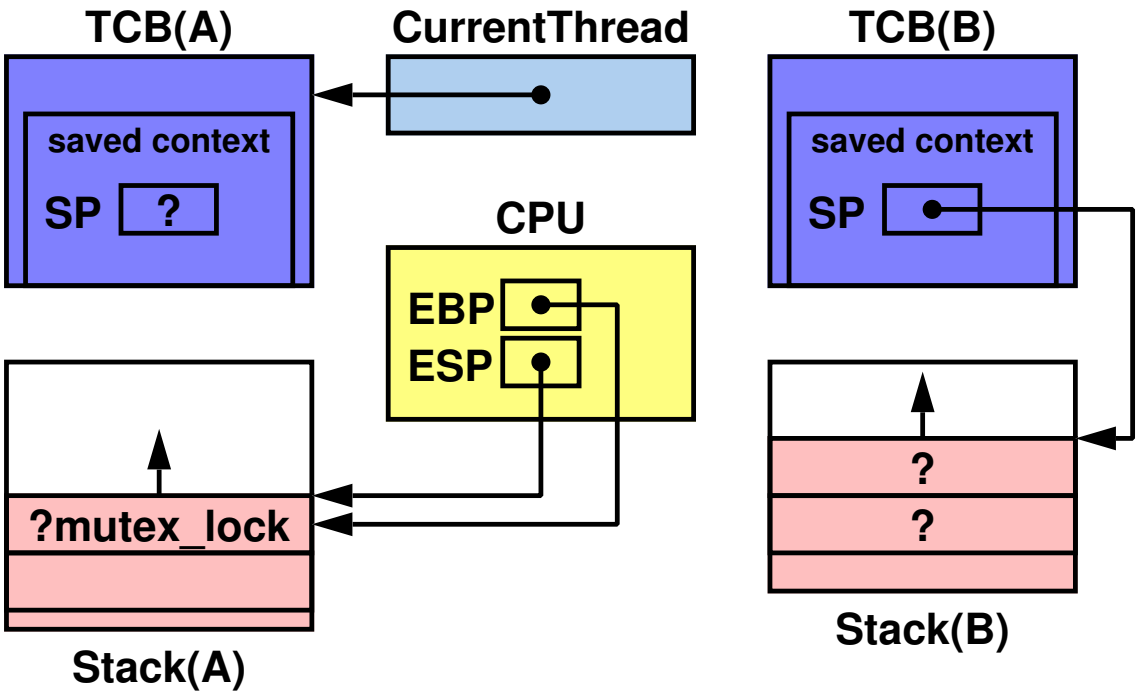


  - to transfer control from one thread to another is equivalent to copying the thread control block of the target thread into the *"Current Thread context"*
    - we will look at the single CPU case

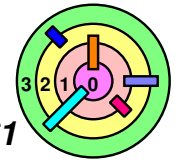


Switching Between Threads

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



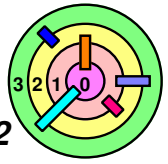
- thread A calls `switch()` to switch to thread B
- thread B's TCB has the *exact context* of thread B right before thread B was *last suspended*
- context information in thread A's TCB is *out-dated*



Switching Between Threads

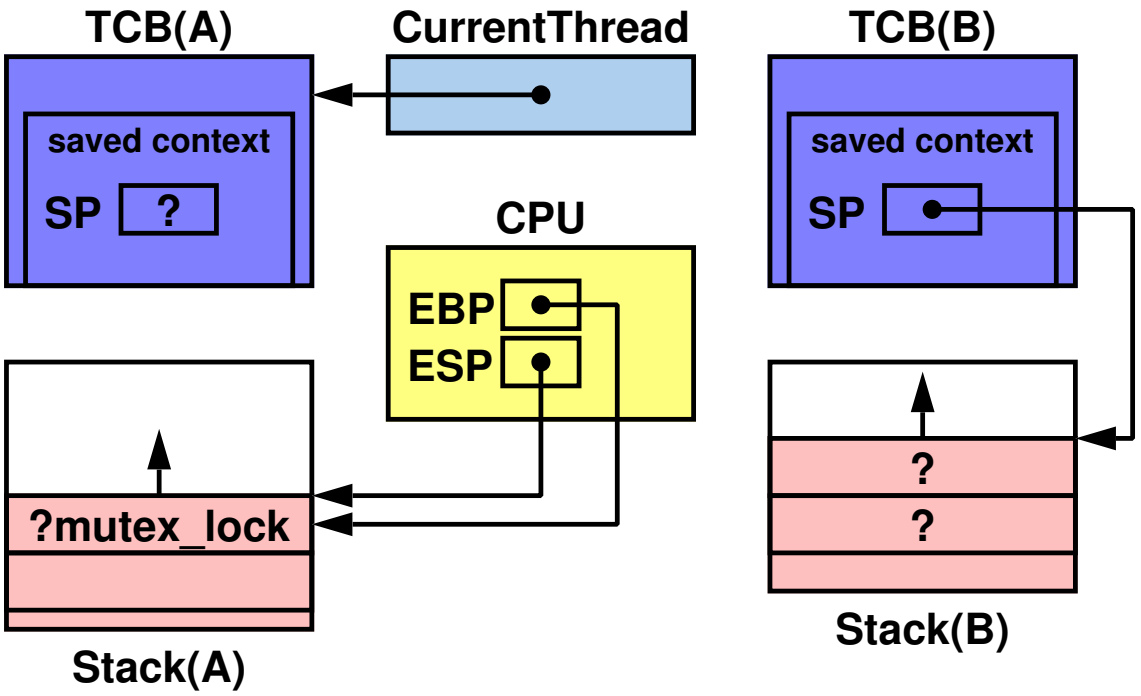
```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```

```
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
```

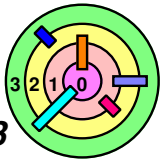


Switching Between Threads

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

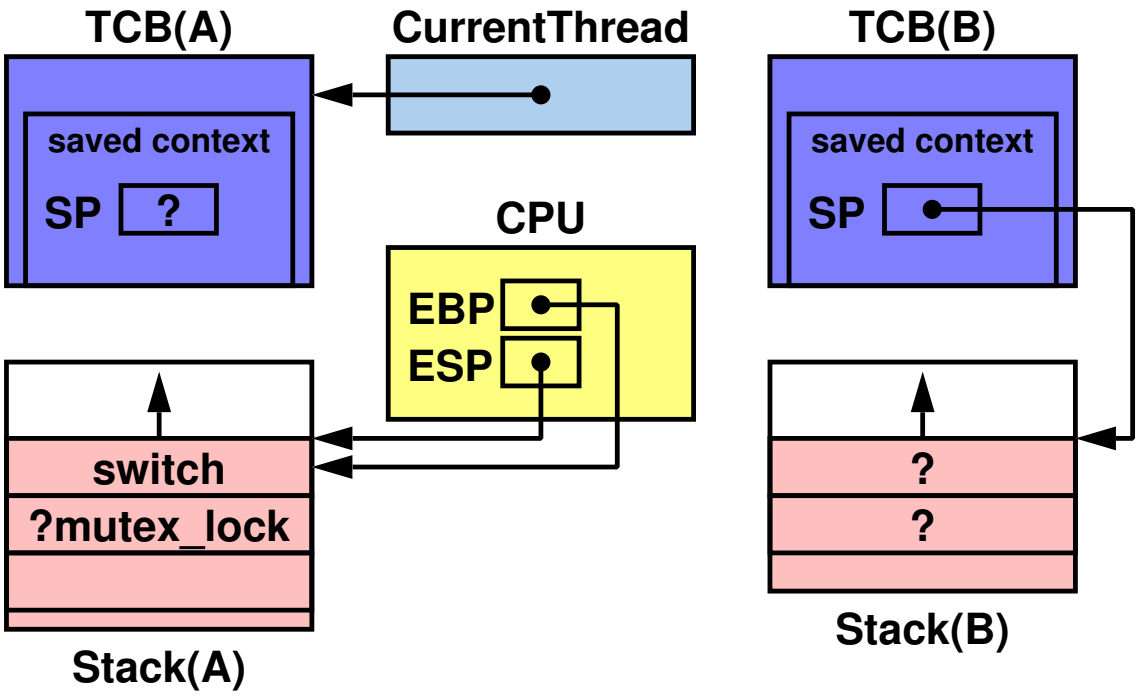


- why is `switch()` called?
  - may be because thread A called `pthread_mutex_lock()` and the mutex is not available



Switching Between Threads

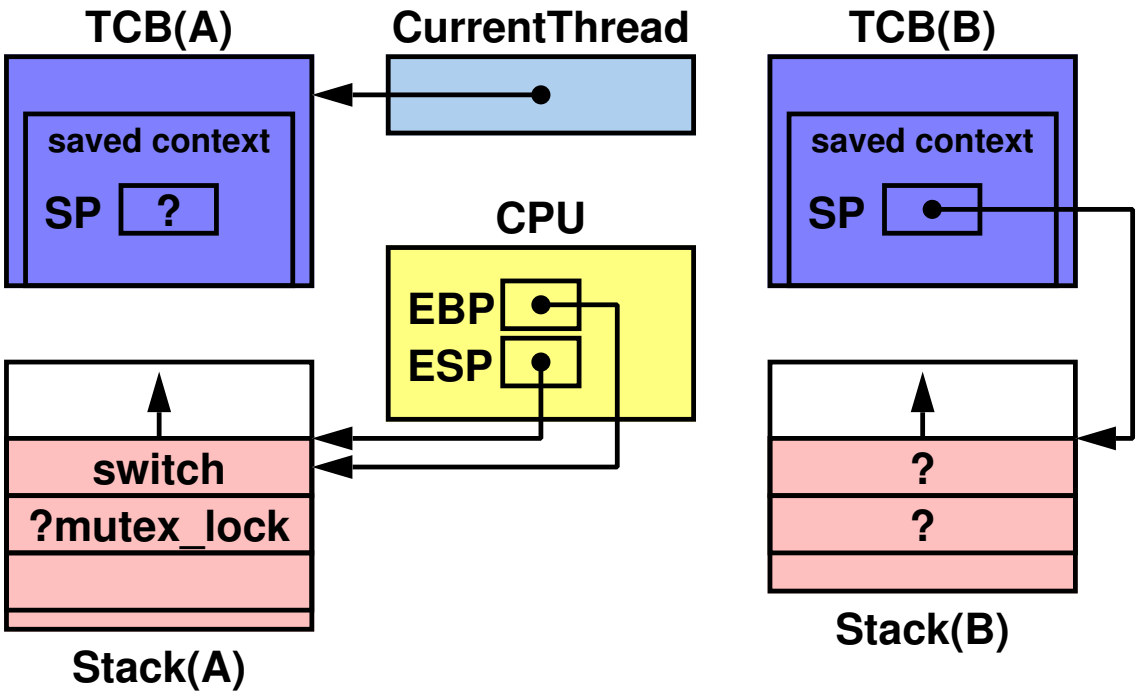
```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```



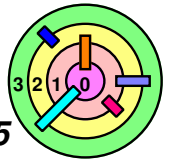
- on entry into `switch()`, the caller's registers are saved!

Switching Between Threads

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

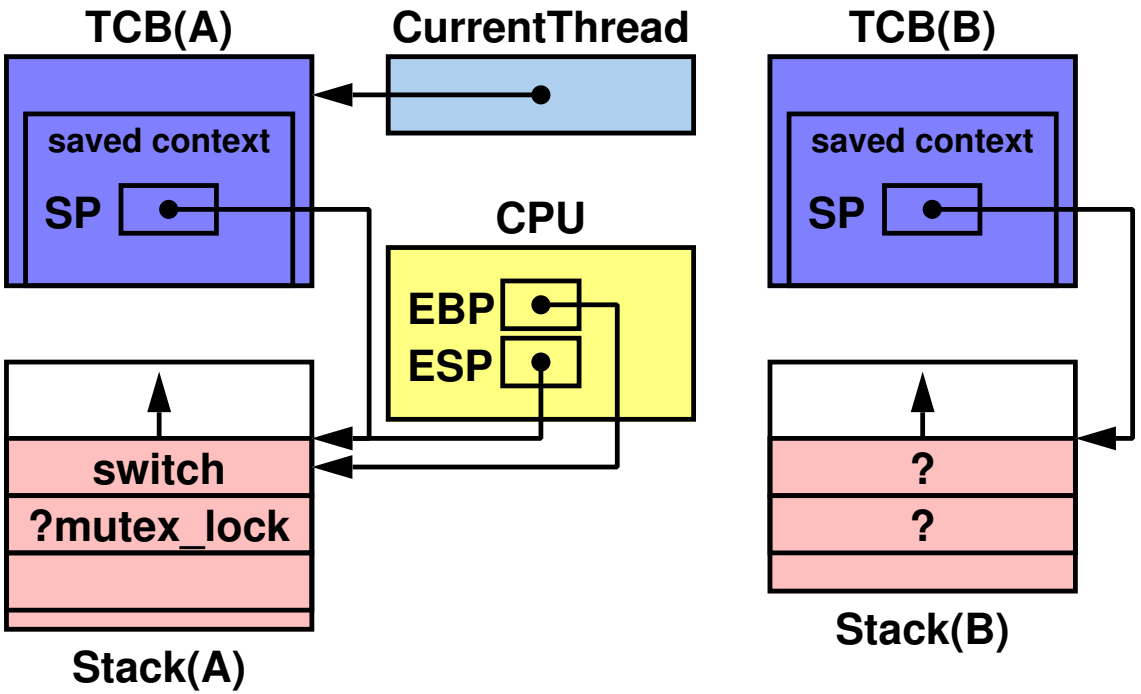


- then the current stack pointer is saved into current thread's thread control block

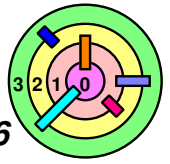


Switching Between Threads

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

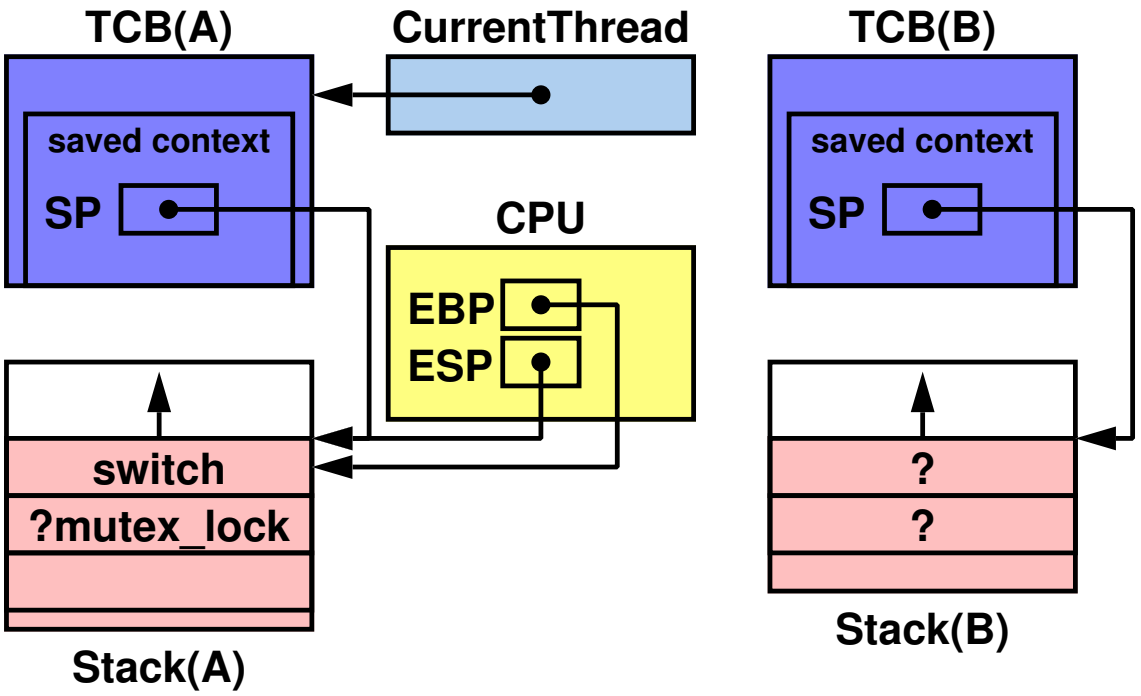


- then the current stack pointer is saved into current thread's thread control block

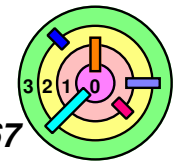


Switching Between Threads

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

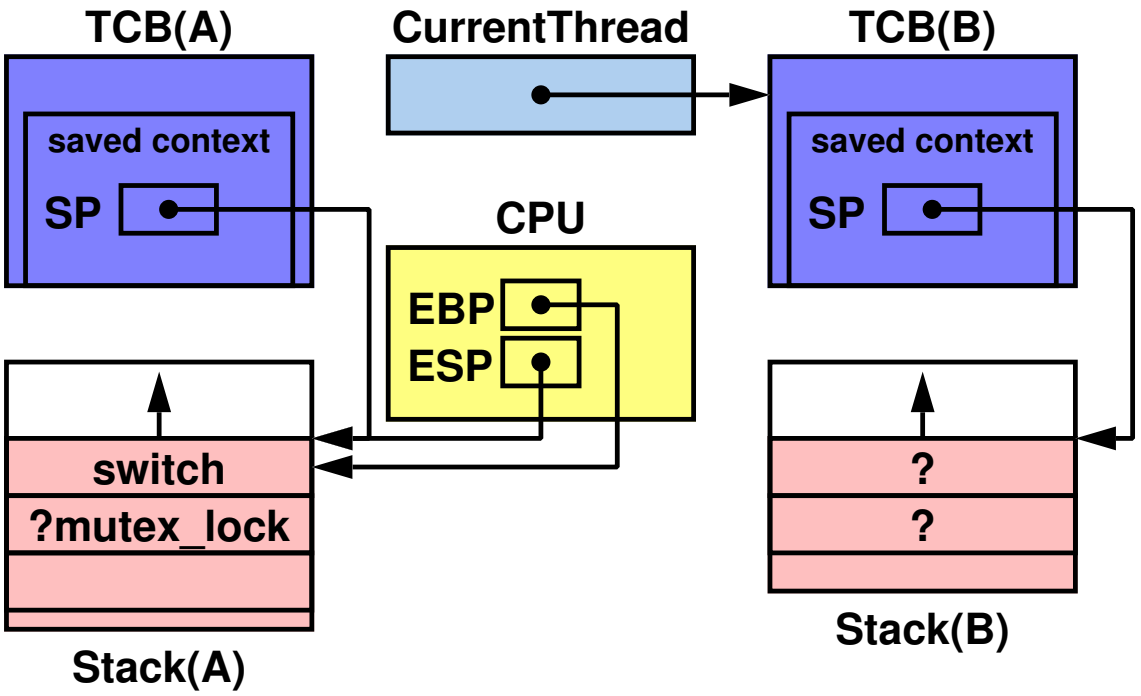


- the target thread becomes the current thread



Switching Between Threads

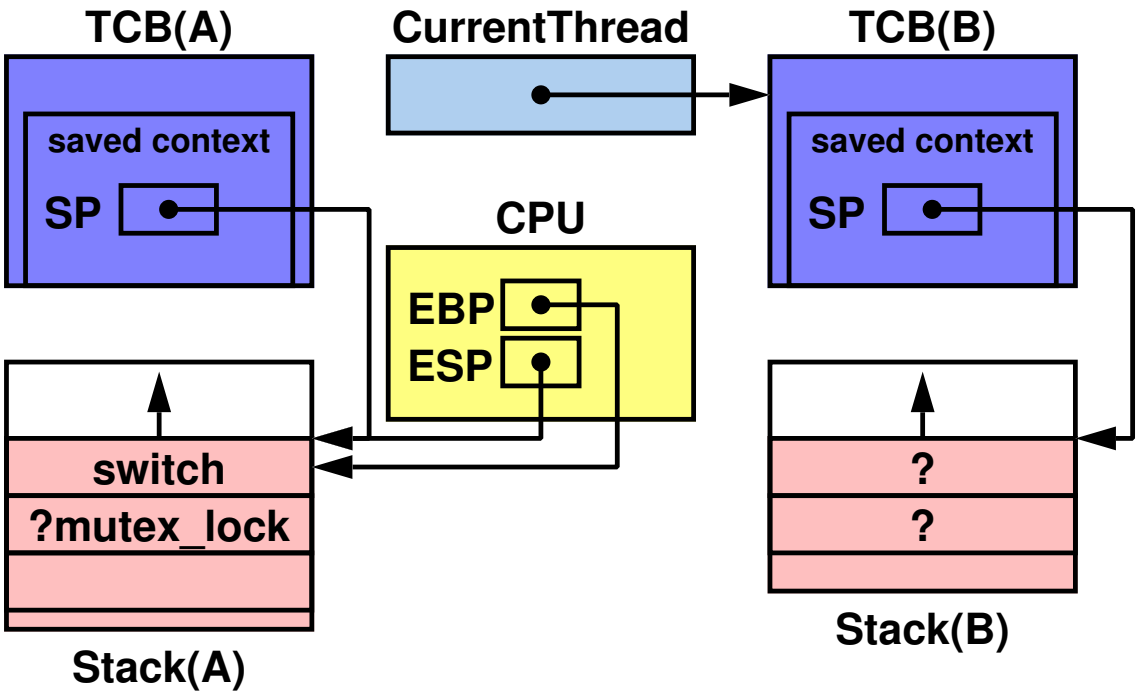
```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```



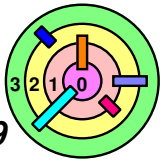
- the target thread becomes the current thread

Switching Between Threads

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

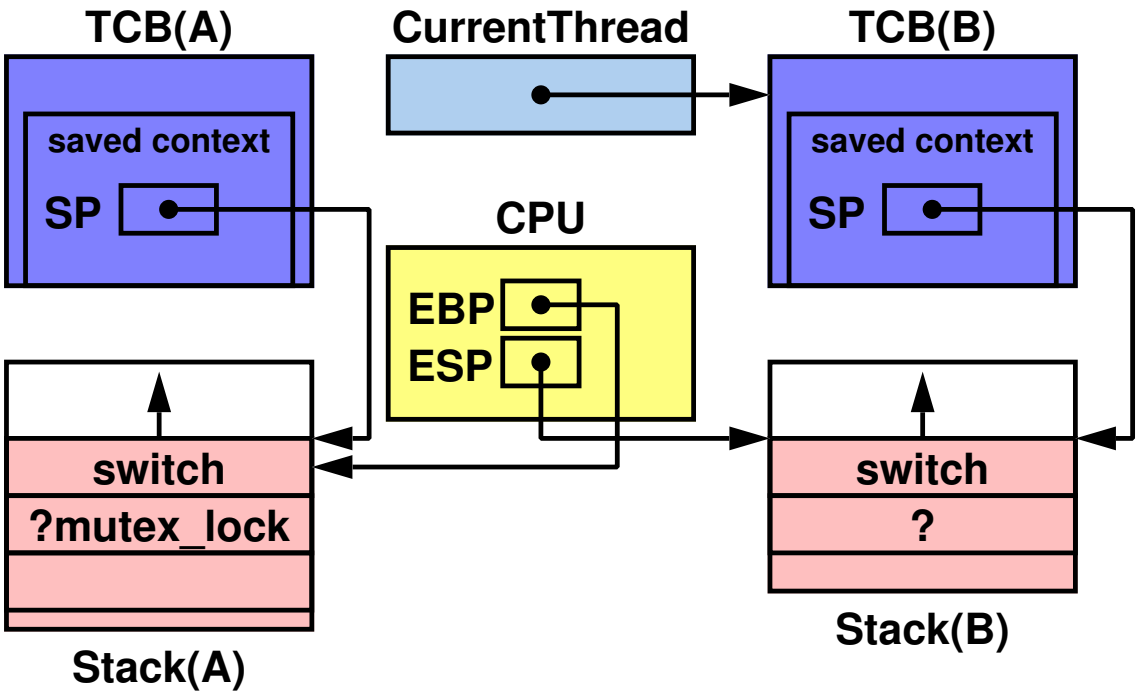


- fetch the target thread's stack pointer (esp for x86) from its thread control block and loads it into the actual stack pointer

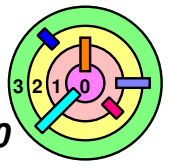


Switching Between Threads

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

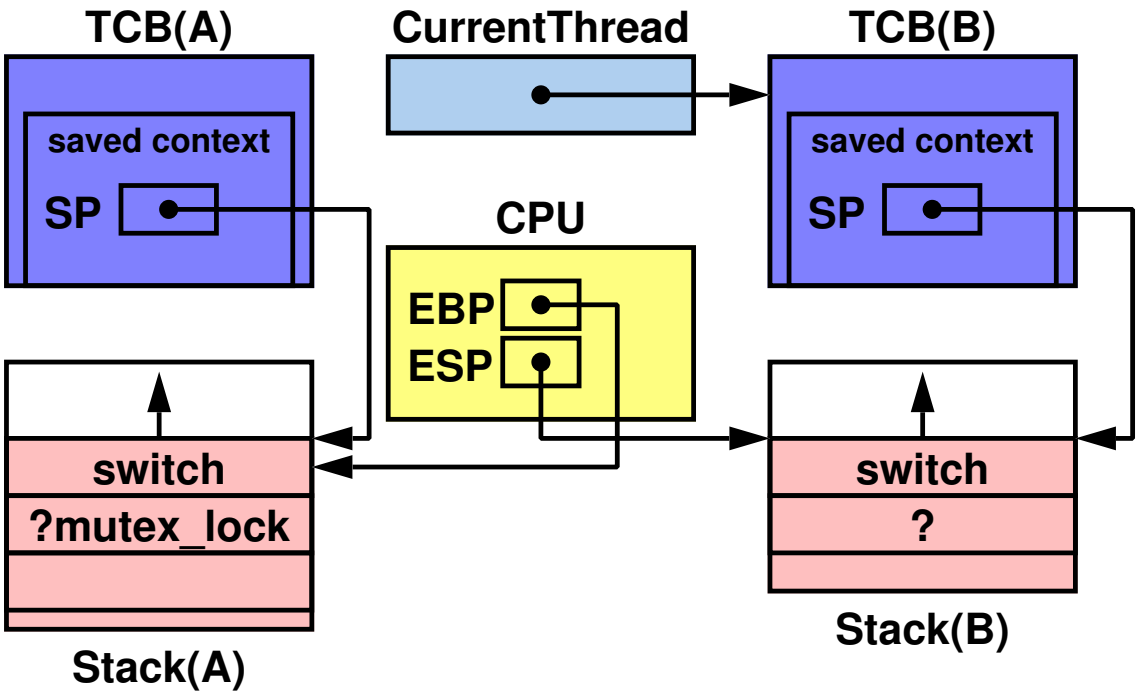


- fetch the target thread's stack pointer (esp for x86) from its thread control block and loads it into the actual stack pointer
  - hmm... which thread executes this?
    - does it matter? what about EIP?



Switching Between Threads

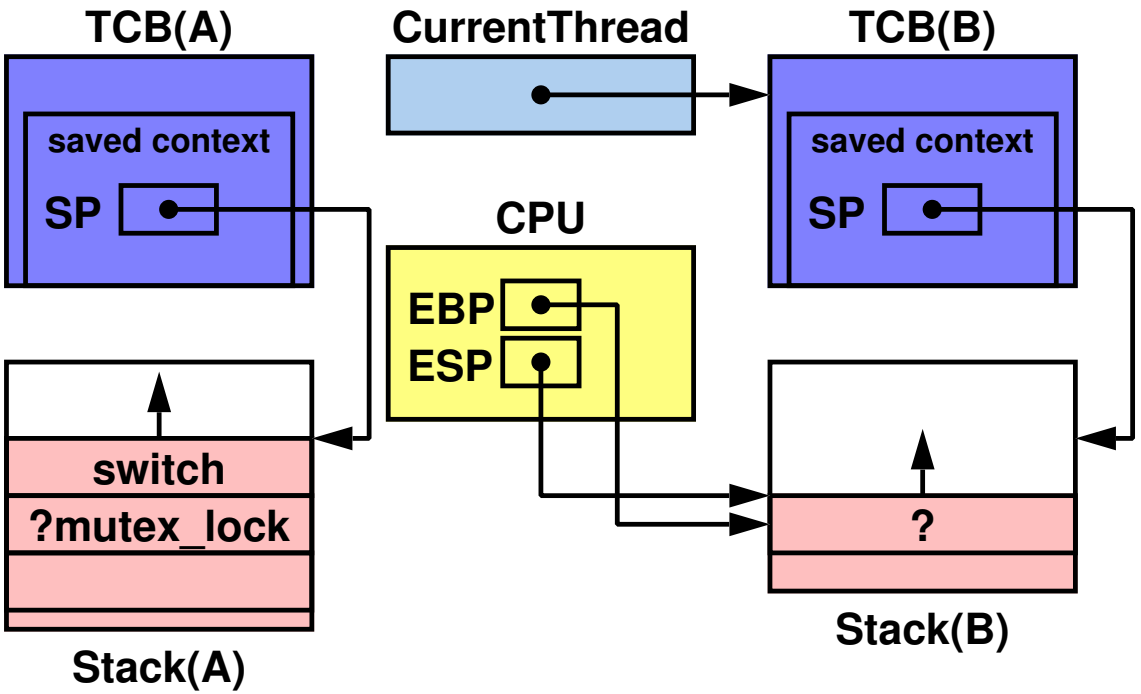
```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```



- 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`?
    - does it matter? what about EIP?

Switching Between Threads

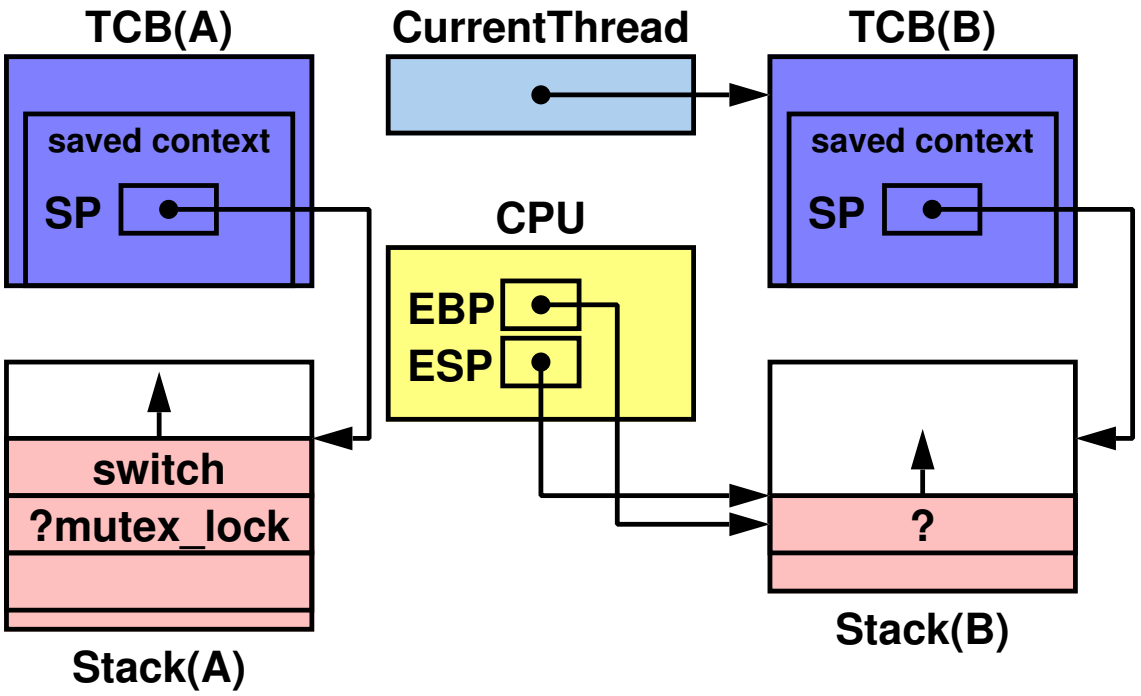
```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```



- 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`?
    - does it matter? what about EIP?

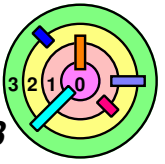
Switching Between Threads

```
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!

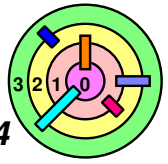
⇨ Note: SP field inside TCB(B) *no longer* tracks ESP in CPU



Switching Between Threads

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

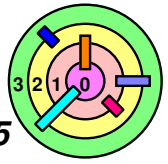
- Note: one very interesting thing happened in this call
  - usually, a single thread executes the entire procedure call
  - with `switch()`, at the beginning of the procedure call, one thread is executing
    - half way through the procedure call, another thread starts to execute
    - so, one thread enters the `switch()` call, and *a different thread* leaves the `switch()` call!
    - but from whose perspective?!



Switching Between Threads

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

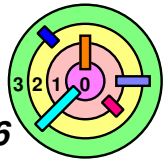
- With `switch()`, one thread enters the `switch()` call, and *a different thread* leaves the `switch()` call!
  - that's from the *system's perspective*
  - from the *calling thread's perspective*, it fell asleep in `switch()`
  - from the *next_thread's perspective*, it woke up in `switch()`
    - it fell asleep a while back and now woke up somehow
  - from the *CPU's perspective*
    - "I did what?! What's a thread?"
    - *thread* is just an *abstraction*, CPU doesn't really know about threads



Switching Between Threads

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

- This is an elegant way of switching threads
  - all threads come here to switch to another thread
- This code is incomplete
  - what if `next_thread` is `NULL`?
  - it saves *some* of the CPU registers, but we need to save *all* of the CPU registers
  - will revisit in Ch 5

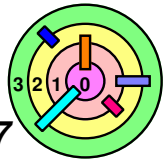


... in x86 Assembler

```
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
```

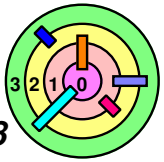
```
void switch(thread_t *next_thread) {
  CurrentThread->SP = SP;
  CurrentThread = next_thread;
  SP = CurrentThread->SP;
  return;
}
```

- The above is ***hand-written*** and not compiler generated
  - can easily change it to save more registers



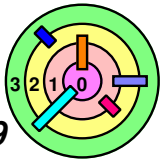
... in SPARC Assembler

```
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).
```



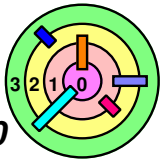
3.1 Context Switching

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



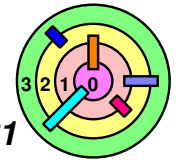
System Calls

- A system call involves the transfer of control from user code to system/kernel code and back
  - there is *no thread switching!*
    - although there was a *context switch*
  - depending on the OS implementation (and as far as this class is concerned), 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
  - it then changed back to a user thread eventually



System Calls

- Most systems provide threads with *two stacks*
  - one for use in user mode
  - and one for use in kernel mode (as far as this class is concerned)
    - in some systems (not common these days), one kernel stack is shared by all threads in the same user process (Ch 5)
  - 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
      - what if the user process has almost used up the stack space?

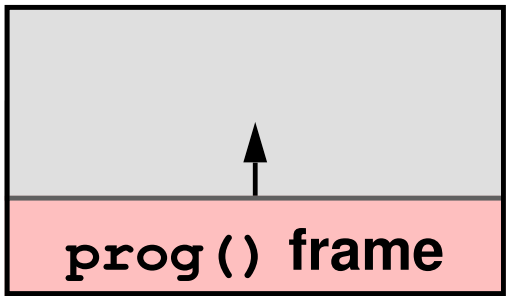


System Calls

- 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

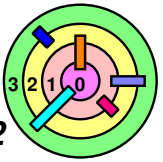
User

Kernel

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

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

Kernel Stack

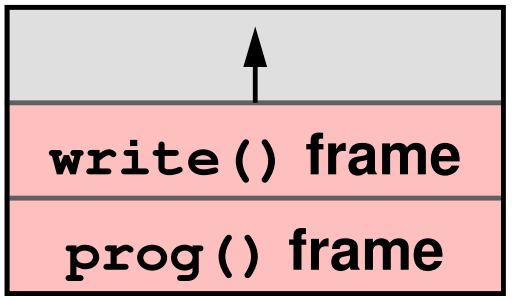


System Calls

- 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

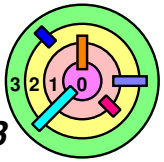
User

Kernel

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

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

Kernel Stack

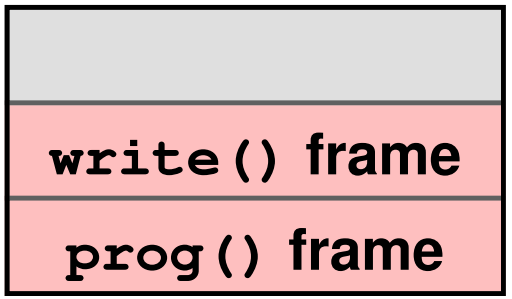


System Calls

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

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

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



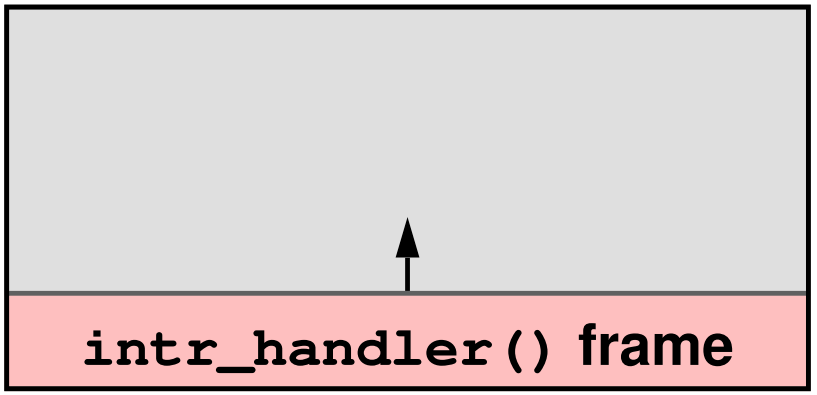
User Stack

User

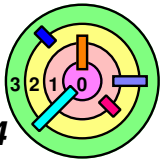
Kernel

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

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



Kernel Stack

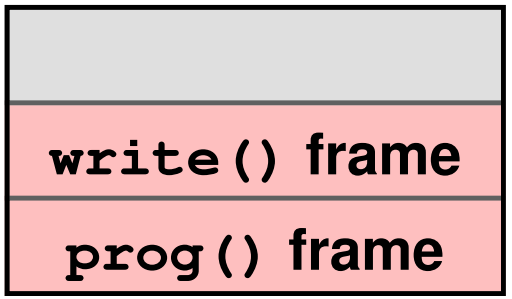


System Calls

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

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

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



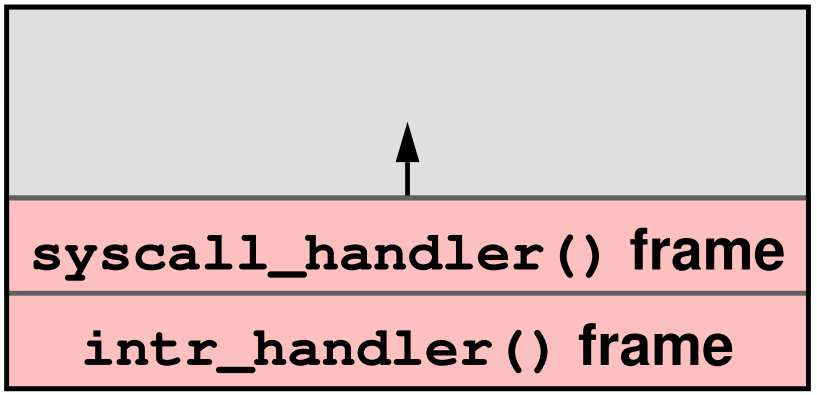
User Stack

User

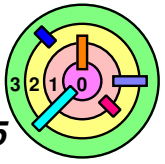
Kernel

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

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



Kernel Stack

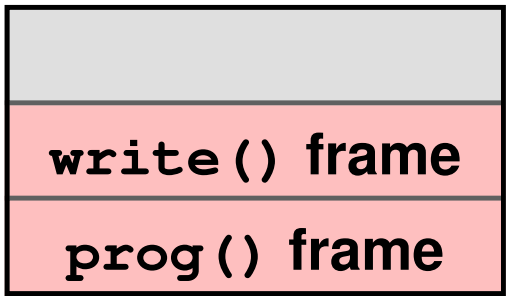


System Calls

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

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

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



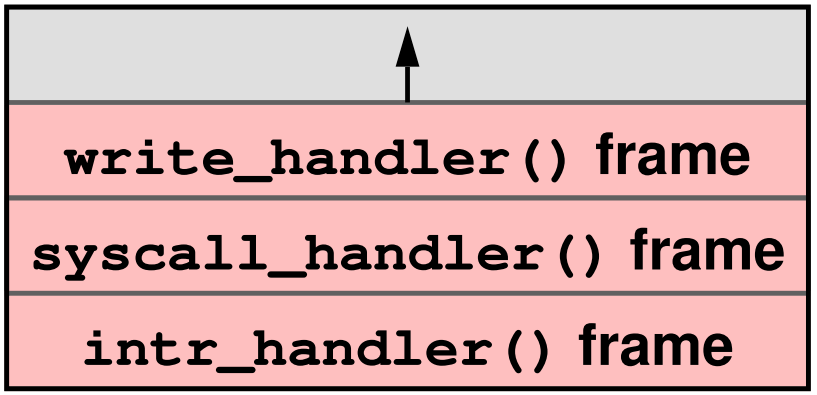
User Stack

User

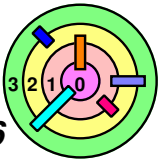
Kernel

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

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



Kernel Stack

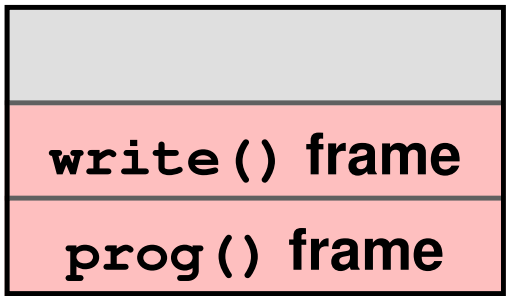


System Calls

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

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

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



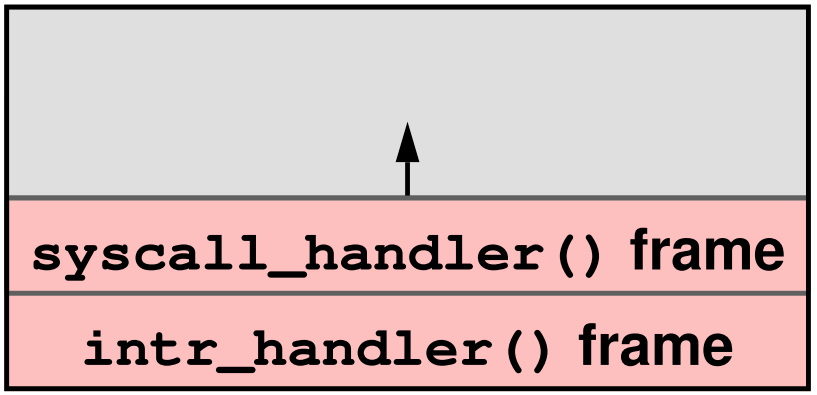
User Stack

User

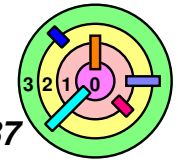
Kernel

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

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



Kernel Stack

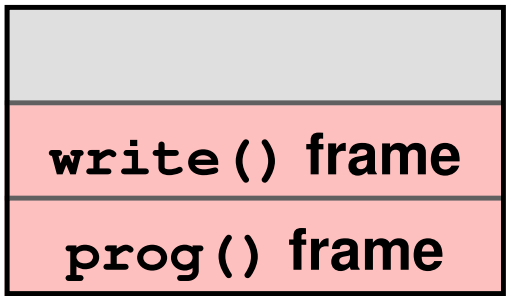


System Calls

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

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

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



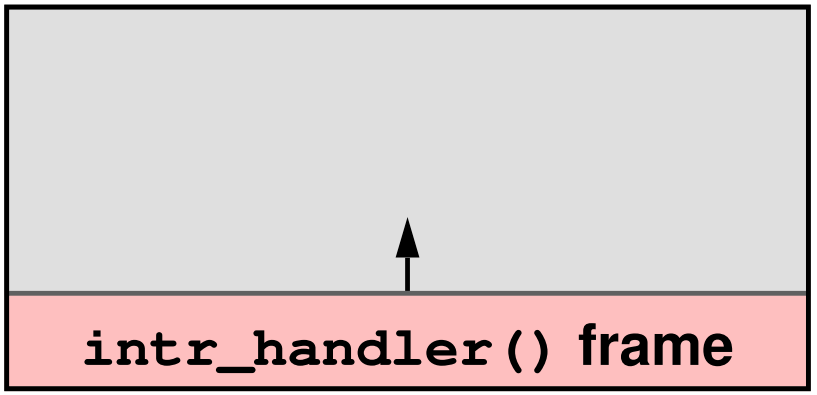
User Stack

User

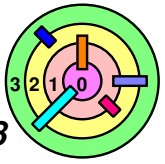
Kernel

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

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



Kernel Stack

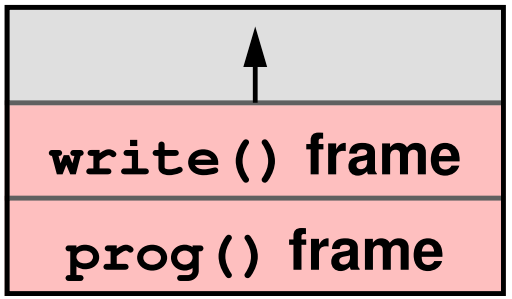


System Calls

- 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

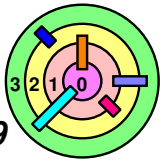
User

Kernel

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

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

Kernel Stack

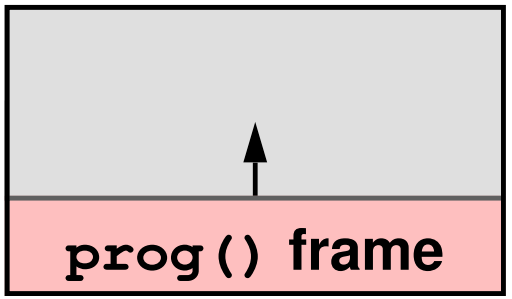


System Calls

- 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