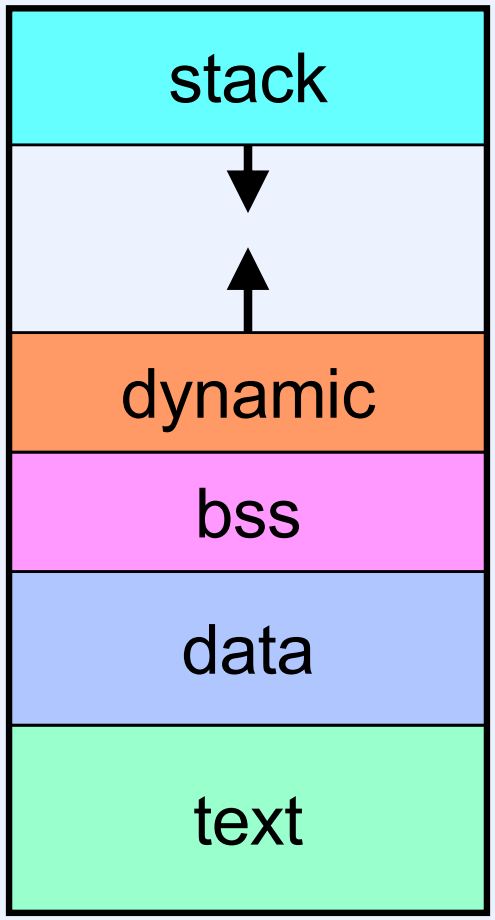
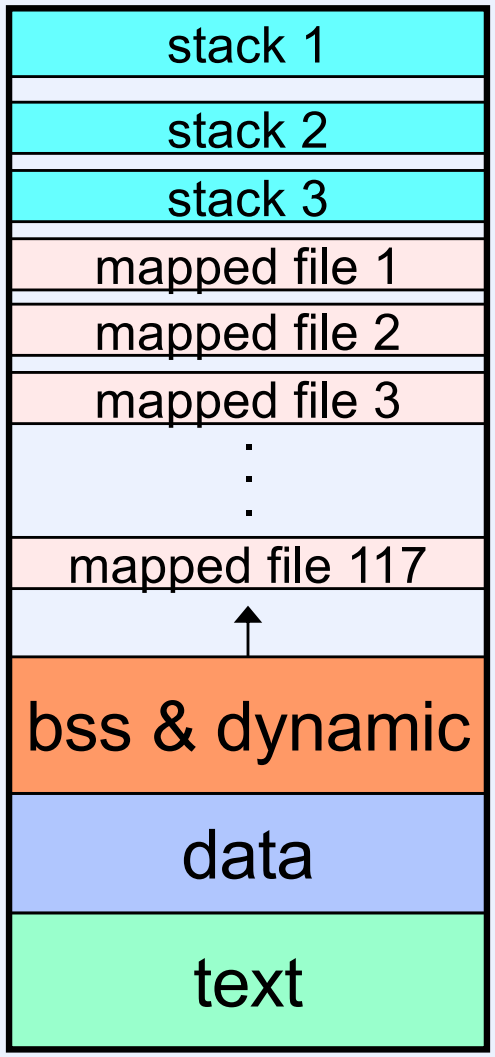
Implementing Threads

The Unix Address Space



Adding More Stuff

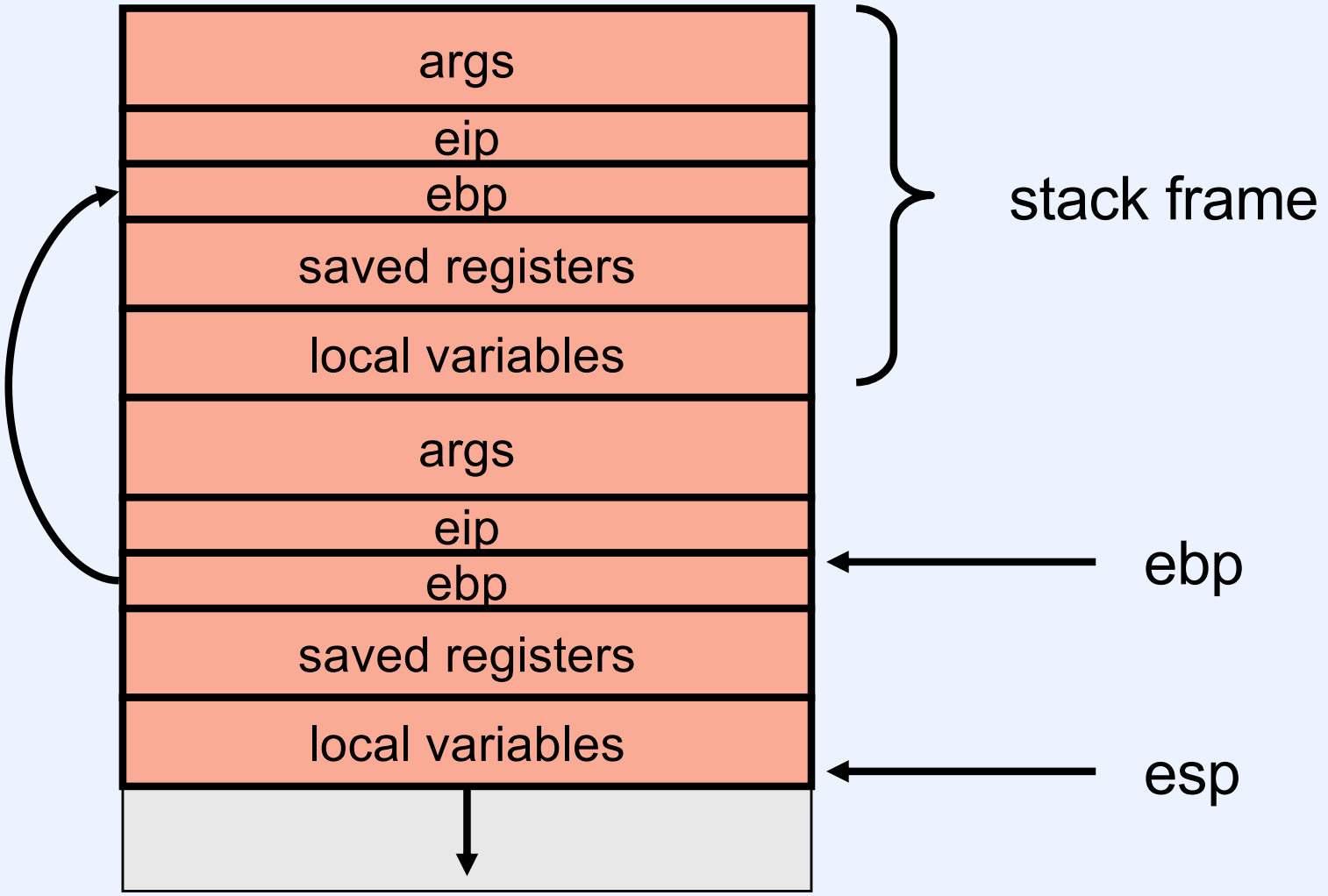


Subroutines

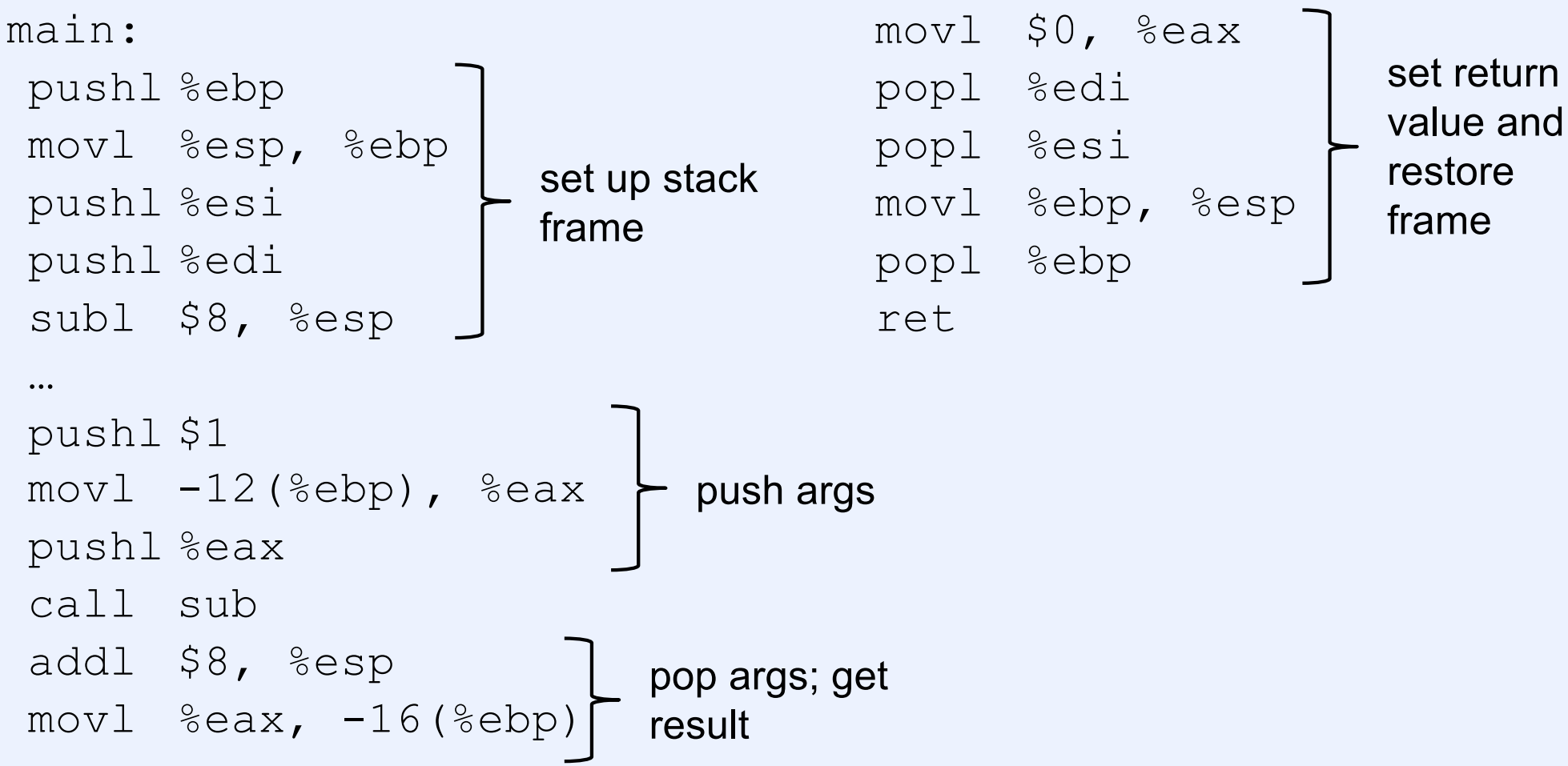
```
int main( ) {
    int i;
    int a;

    ...

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

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

Intel x86 (32-Bit):
Subroutine Linkage



Intel x86: Subroutine Code (1)



Intel x86:
Subroutine Code (2)

```
sub:
 pushl %ebp
 movl  %esp, %ebp
 subl  $8, %esp
 movl  $1, -4(%ebp)        initialize result
 movl  $0, -8(%ebp)        initialize i
 movl  -4(%ebp), %ecx      %ecx holds result
 movl  -8(%ebp), %eax      %eax holds  i
beginloop:
 cmpl  12(%ebp), %eax      y:i
 jge endloop
 imull 8(%ebp), %ecx       result *= x
 addl  $1, %eax            i++
 jmp beginloop
endloop:
 movl  %ecx, -4(%ebp)      result = %ecx
 movl  -4(%ebp), %eax      %eax = result
 movl  %ebp, %esp
 popl  %ebp
 ret
```

x86-64

- Twice as many registers
- Arguments may be passed in registers, rather than on stack
- No special-purpose base pointer
  - use stack pointer instead

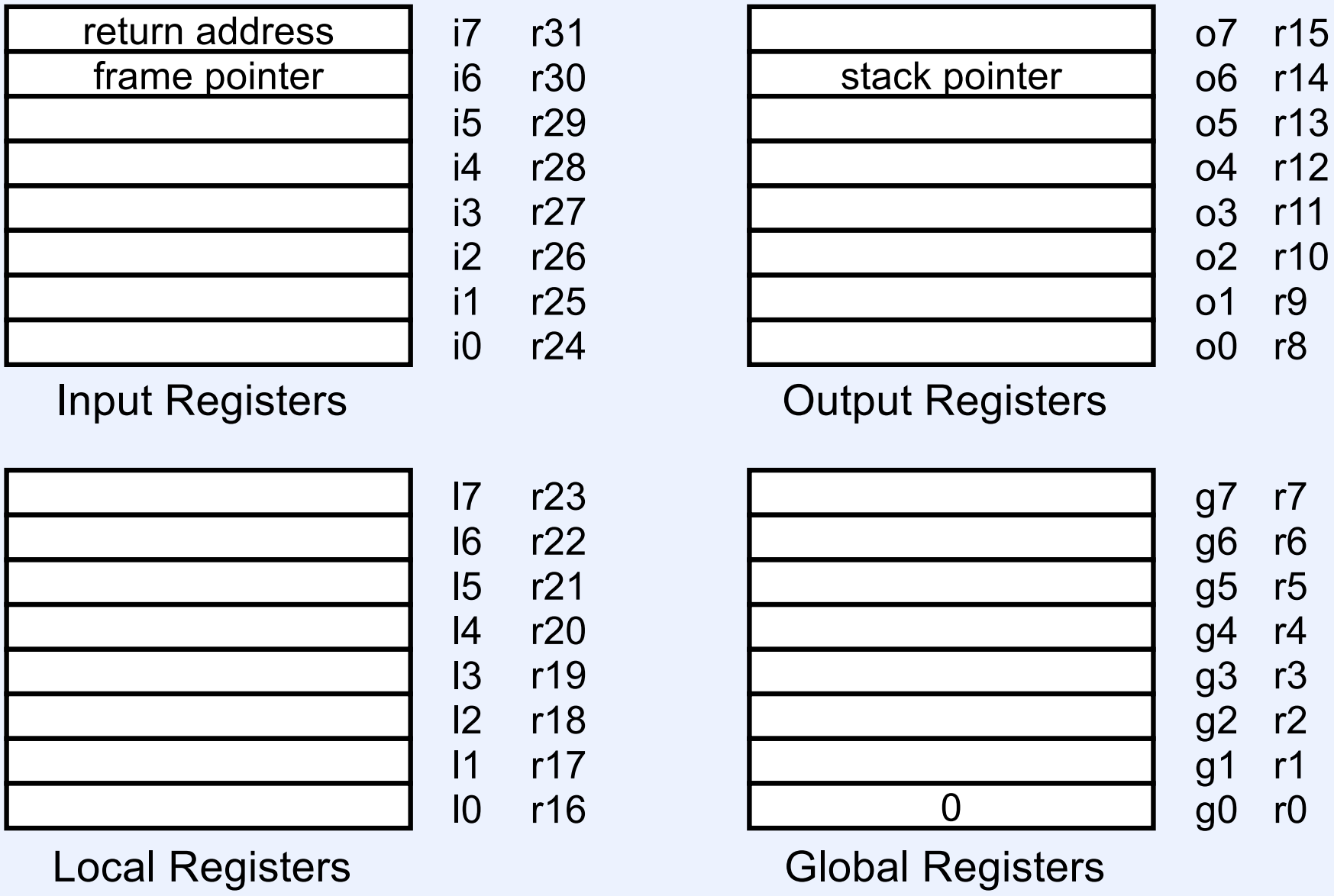
Intel x86-64:
Subroutine Code (1)

```
main:
 subq $24, %rsp          # reserve space on stack for locals
 …
 movl 12(%rsp), %edi     # set first argument
 movl $1, %esi           # set second argument
 call sub
 addl $24, %rsp
 …
 movl $0, %eax           # set return value
 ret
 …
```

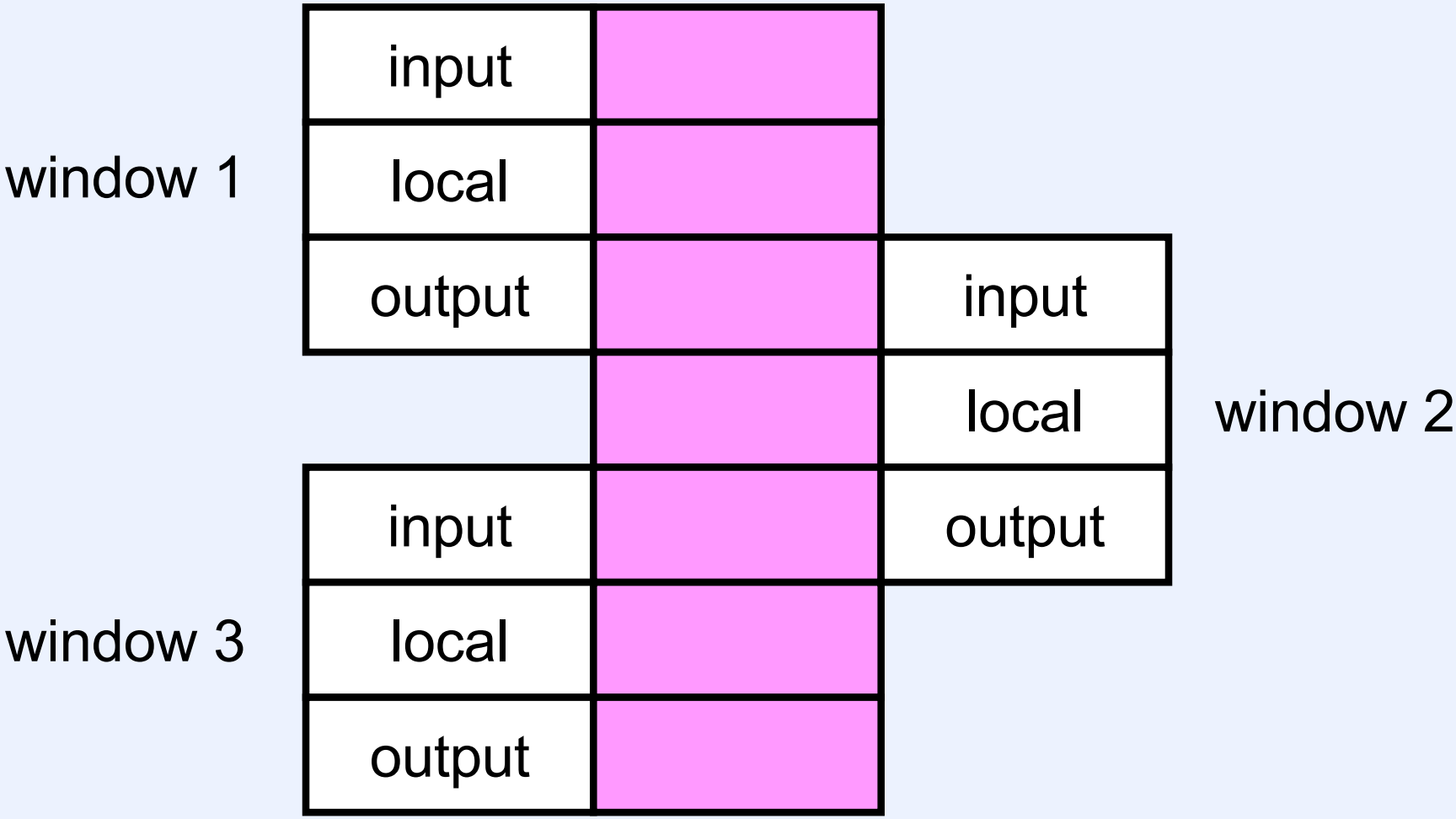
Intel x86-64:
Subroutine Code (2)

```
sub:
 testl %esi, %esi    # leaf function: no stack setup
 jle   skiploop
 movl  $1, %eax
 movl  $0, %edx
loop:
 imull %edi, %eax
 addl  $1, %edx
 cmpl  %esi, %edx
 jne   loop
 ret
skiploop:
 movl  $1, %eax
 ret
```

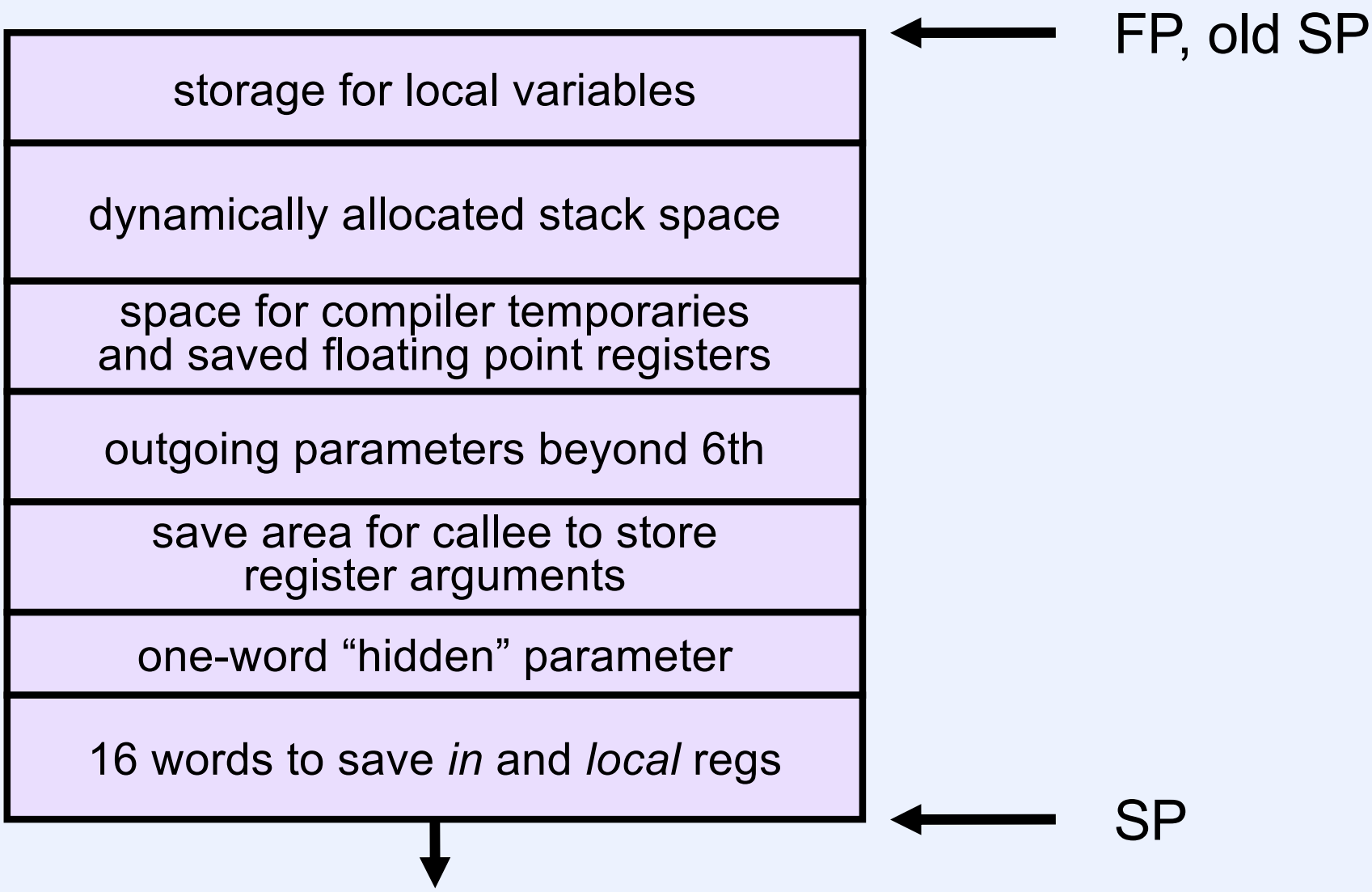
SPARC Architecture



SPARC Architecture:
Register Windows



SPARC Architecture:
Stack

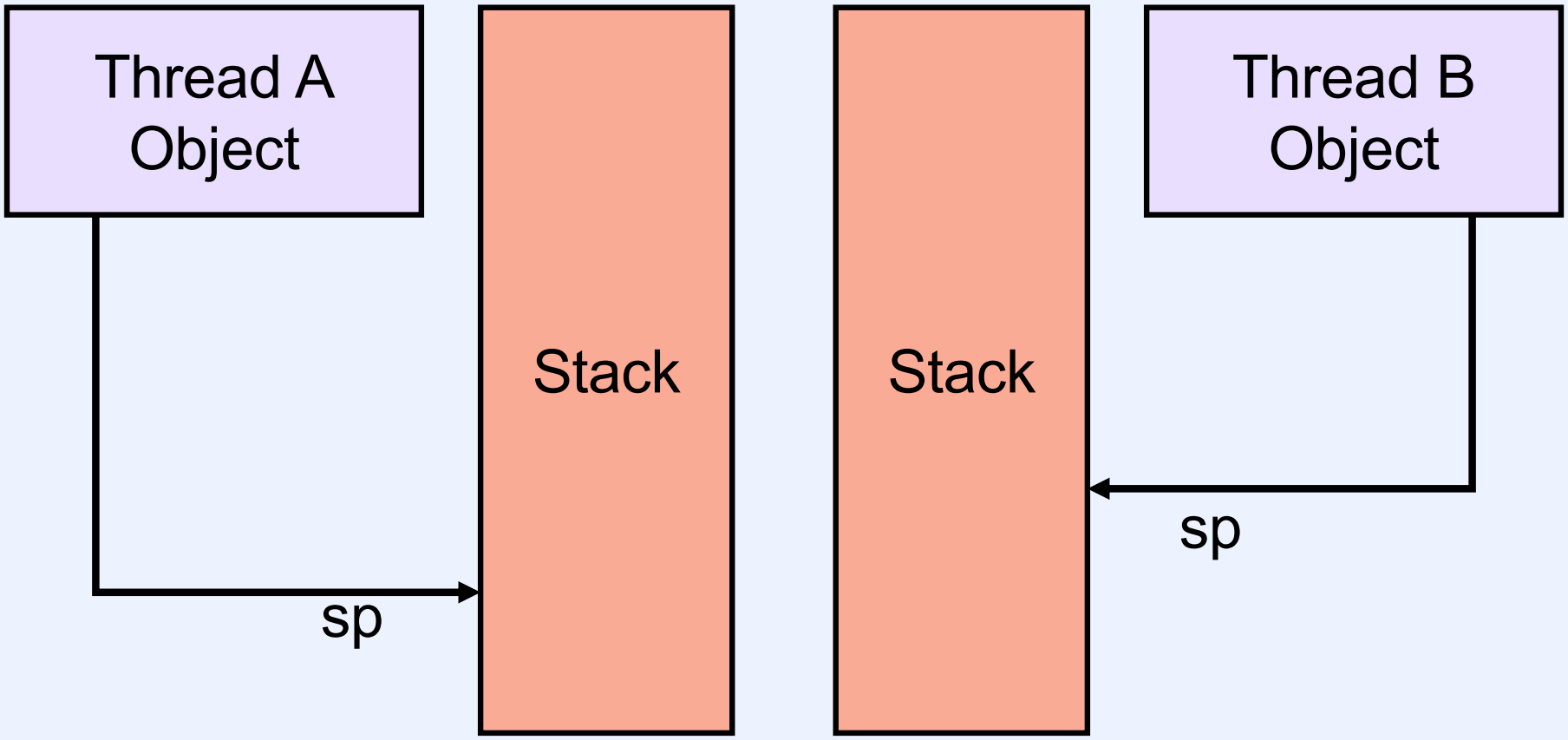


SPARC Architecture:
Subroutine Code

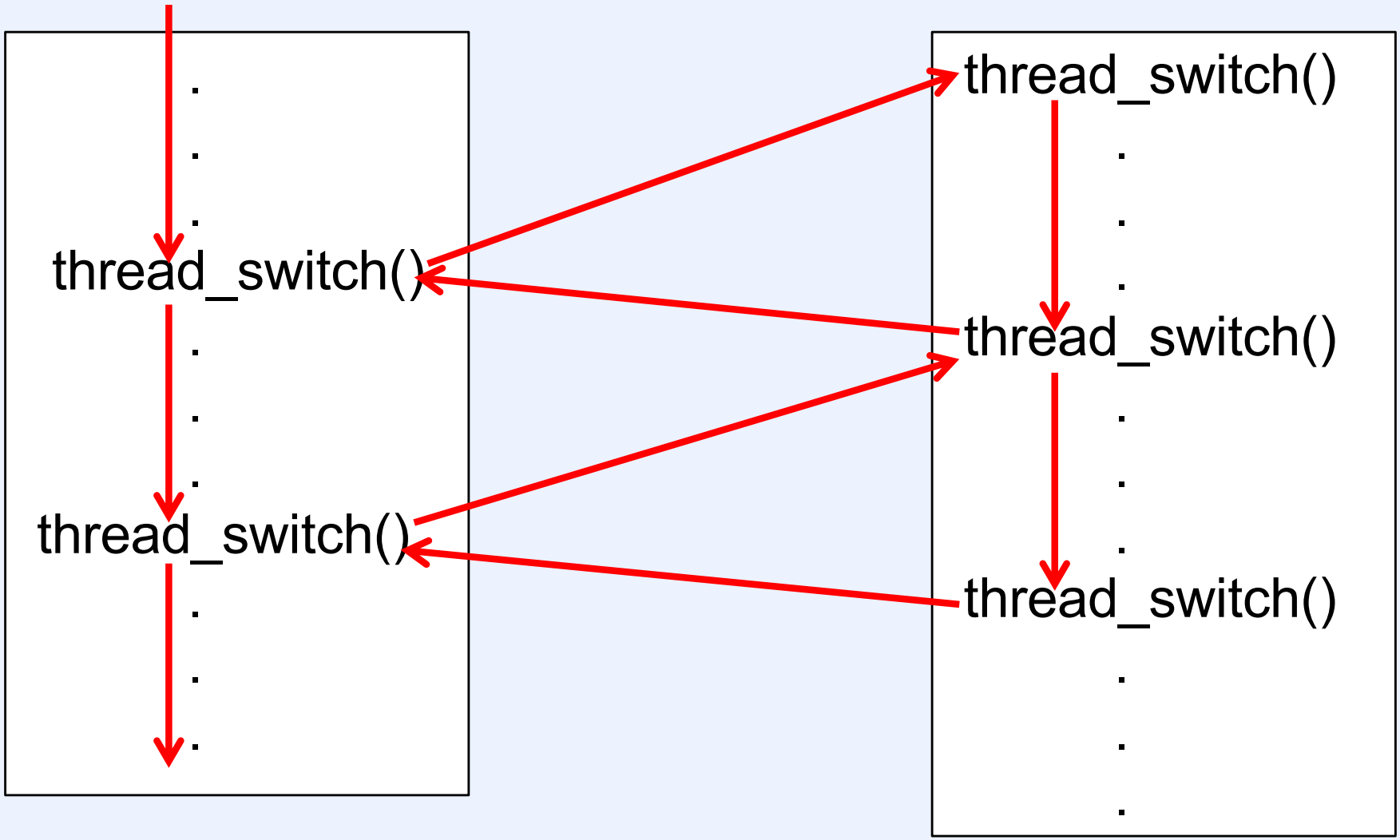
```
ld [%fp-8], %o0
     ! put local var (a)
     ! into out register
mov 1, %o1
     ! deal with 2nd
     ! parameter
call  sub
nop
st    %o0, [%fp-4]
     ! store result into
     ! local var (i)
...
```

```
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)
```

Representing Threads



Switching Between Threads



- **Coroutine linkage**

Switching Between Threads

```
1 void thread_switch(thread_t *next_thread) {
2     getcontext(&CurrentThread->ctx);
3     CurrentThread = next_thread;
4     setcontext(&CurrentThread->ctx);
5     return;
6 }
```

Switching Between Threads, Take 2

```
1   void thread_switch(thread_t *next_thread) {
2       volatile int first = 1;
3       getcontext(&CurrentThread->ctx);
4       if (first) {
5           first = 0;
6           CurrentThread = next_thread;
7           setcontext(&CurrentThread->ctx);
8       }
9       return;
10  }
```

Quiz 1

```
1   void thread_switch(thread_t *next_thread) {
2       volatile int first = 1;
3       getcontext(&CurrentThread->ctx);
4       if (first) {
5           first = 0;
6           CurrentThread = next_thread;
7           setcontext(&CurrentThread->ctx);
8       }
9       return;
10  }
```

Does this implementation of thread_switch work?

a) **yes: in all cases**

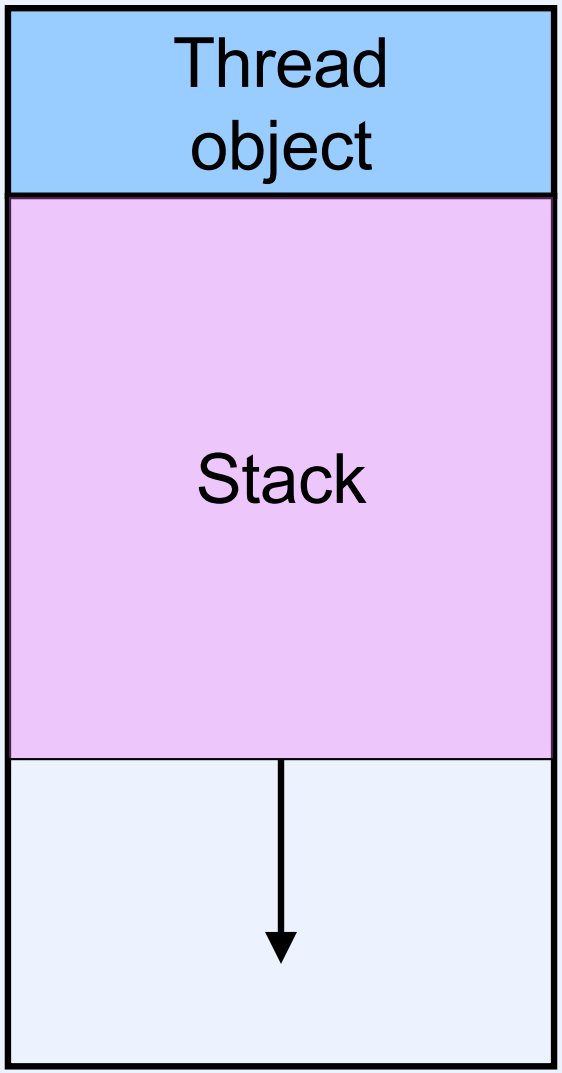
b) **yes, except for a few edge cases**

c) **no**

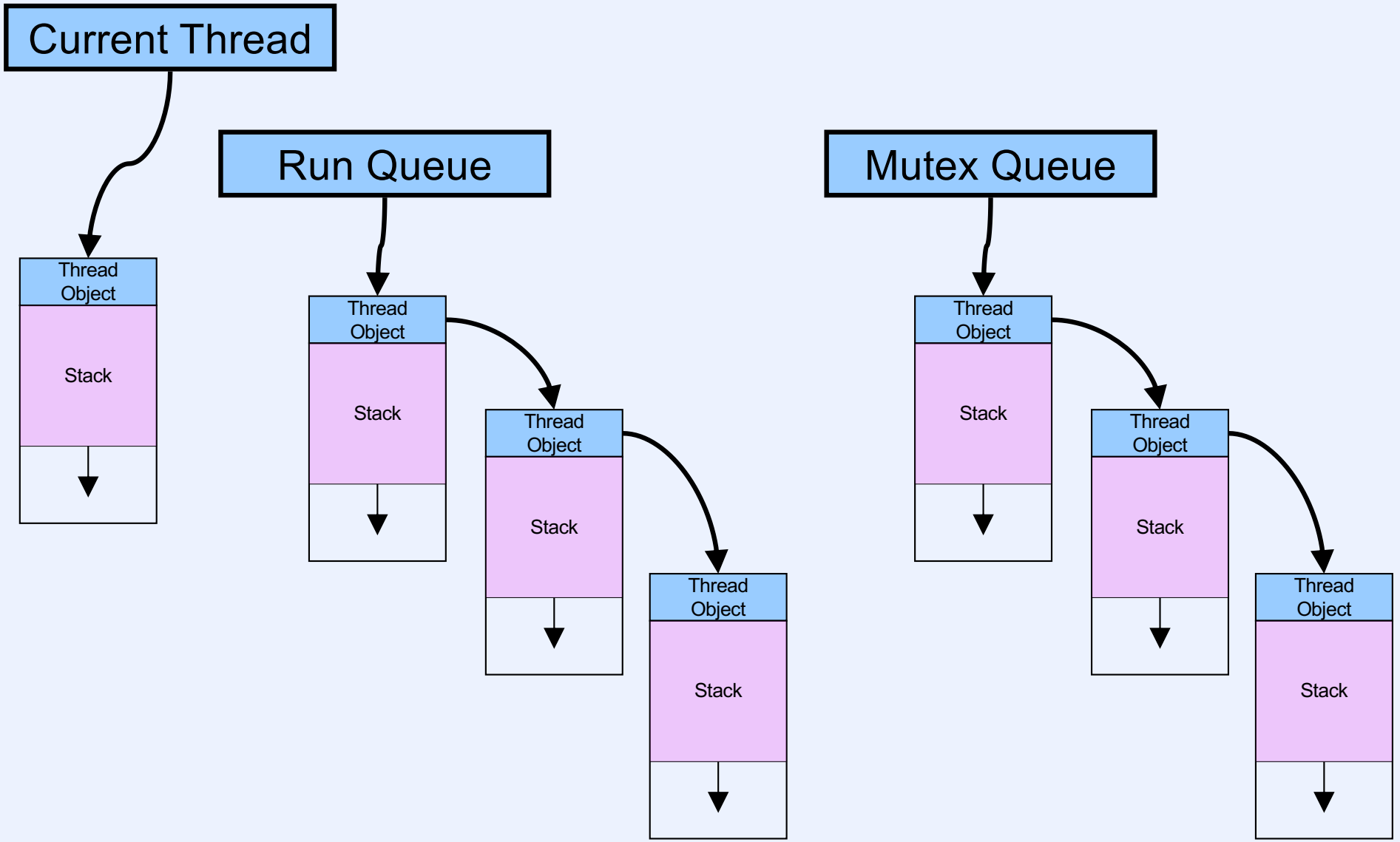
A Simple Threads Implementation

- Basis for user-level threads package
- Straight-threads implementation
  - no interrupts
  - everything in thread contexts
  - one processor

Basic Representation



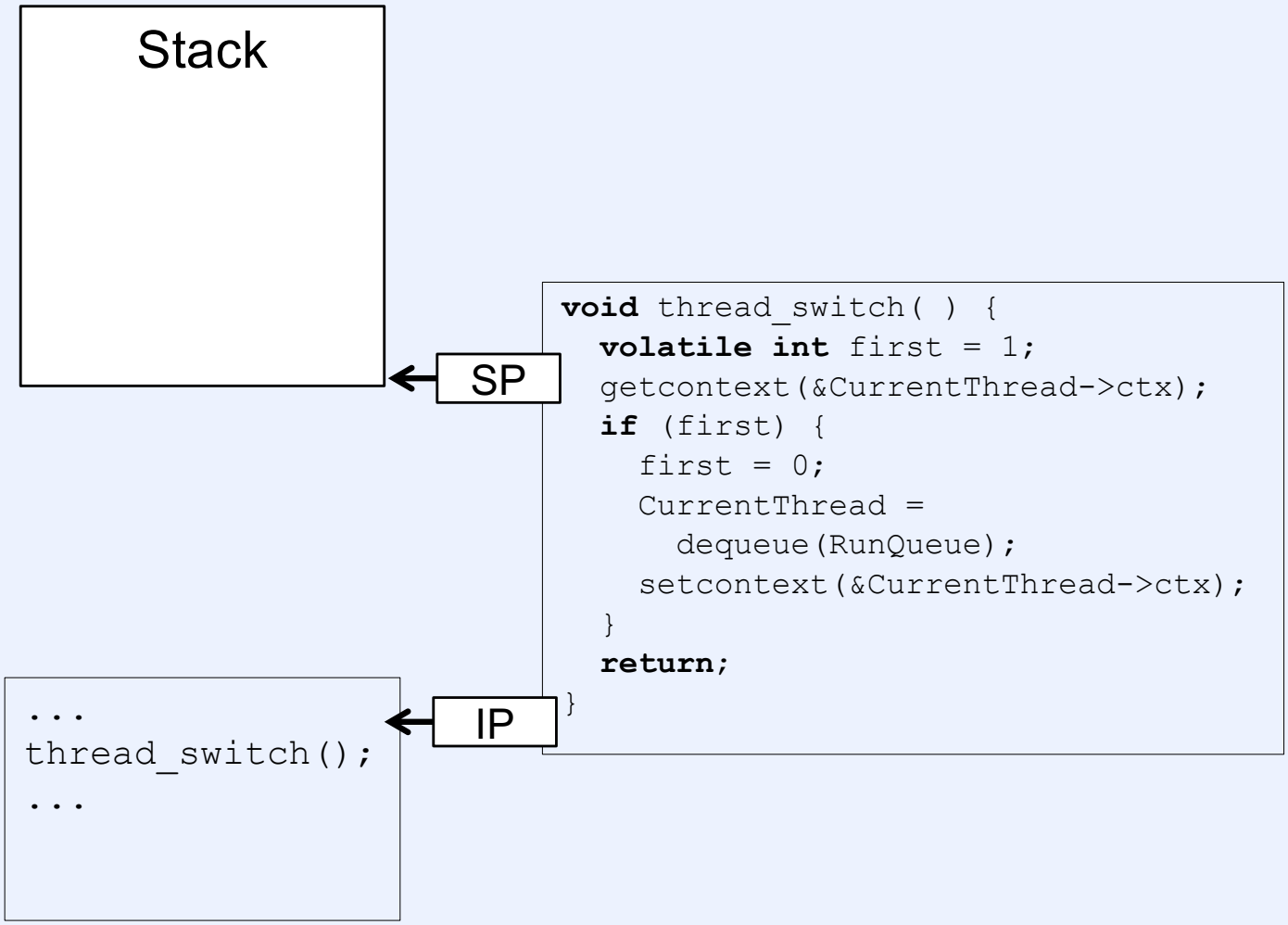
A Collection of Threads



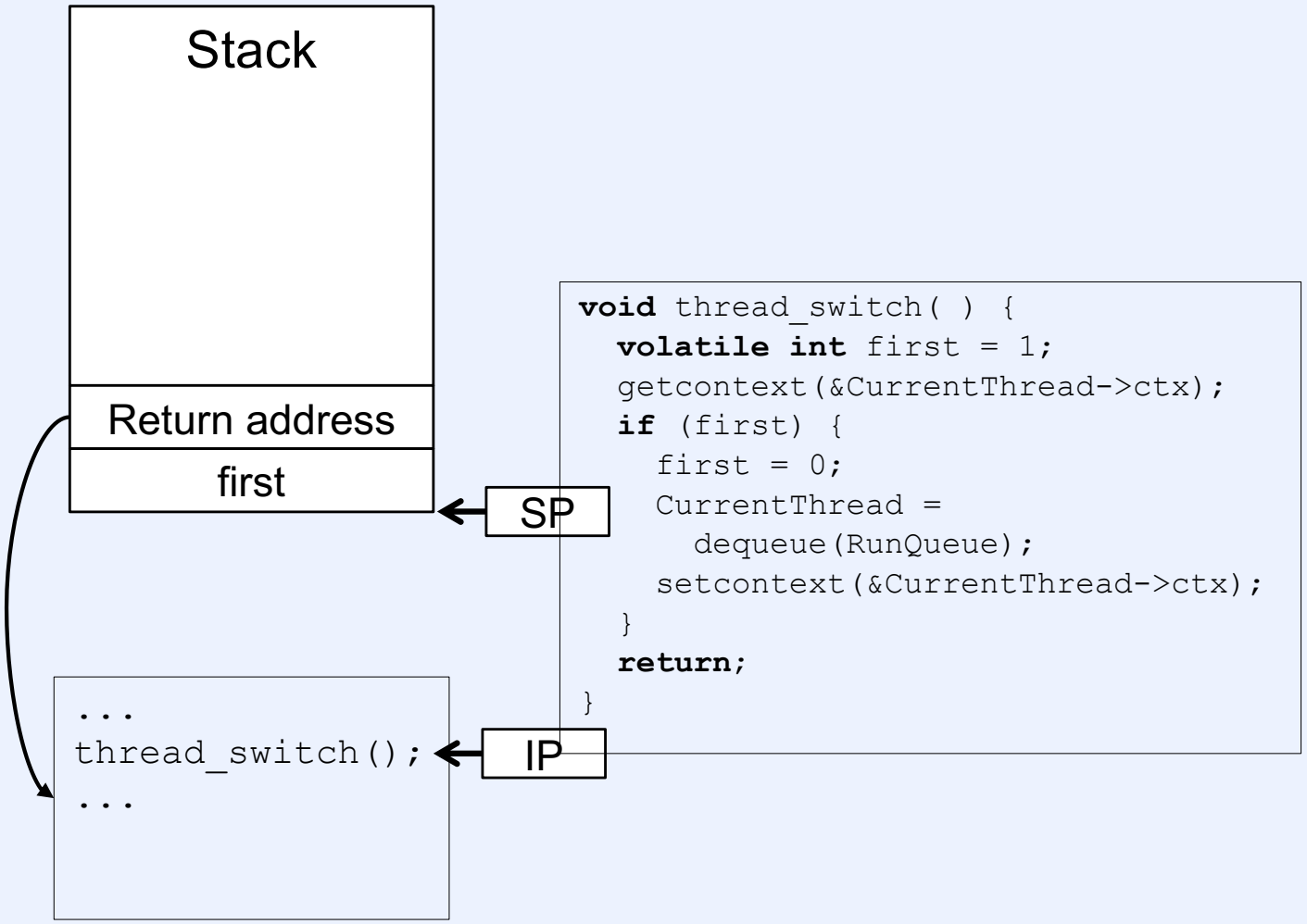
Thread Switch (using Run Queue)

```
void thread_switch( ) {
   volatile int first = 1;
   getcontext(&CurrentThread->ctx);
   if (first) {
      first = 0;
      CurrentThread = dequeue(RunQueue);
      setcontext(&CurrentThread->ctx);
   }
   return;
}
```

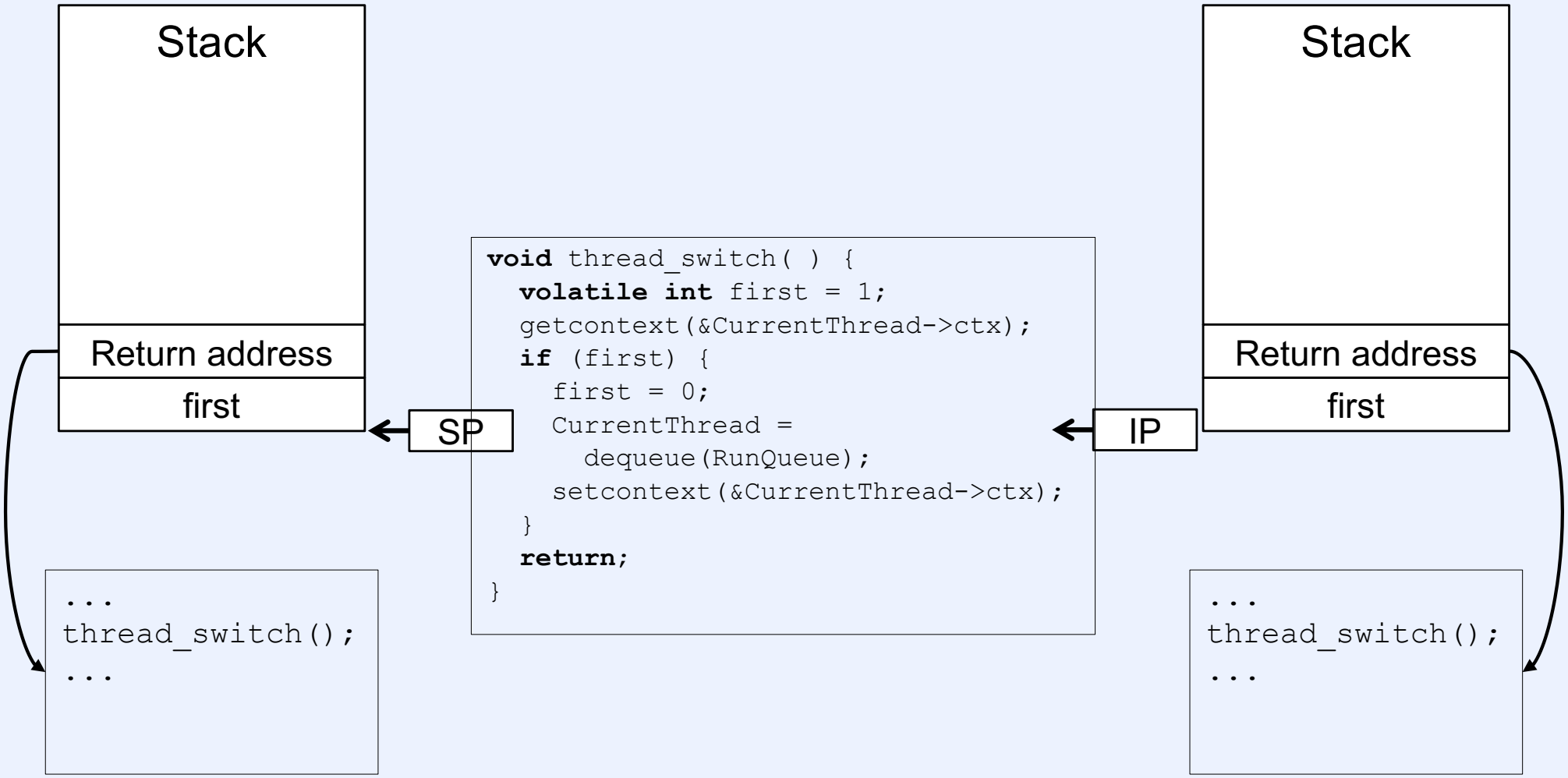
Thread-Switch Exchange



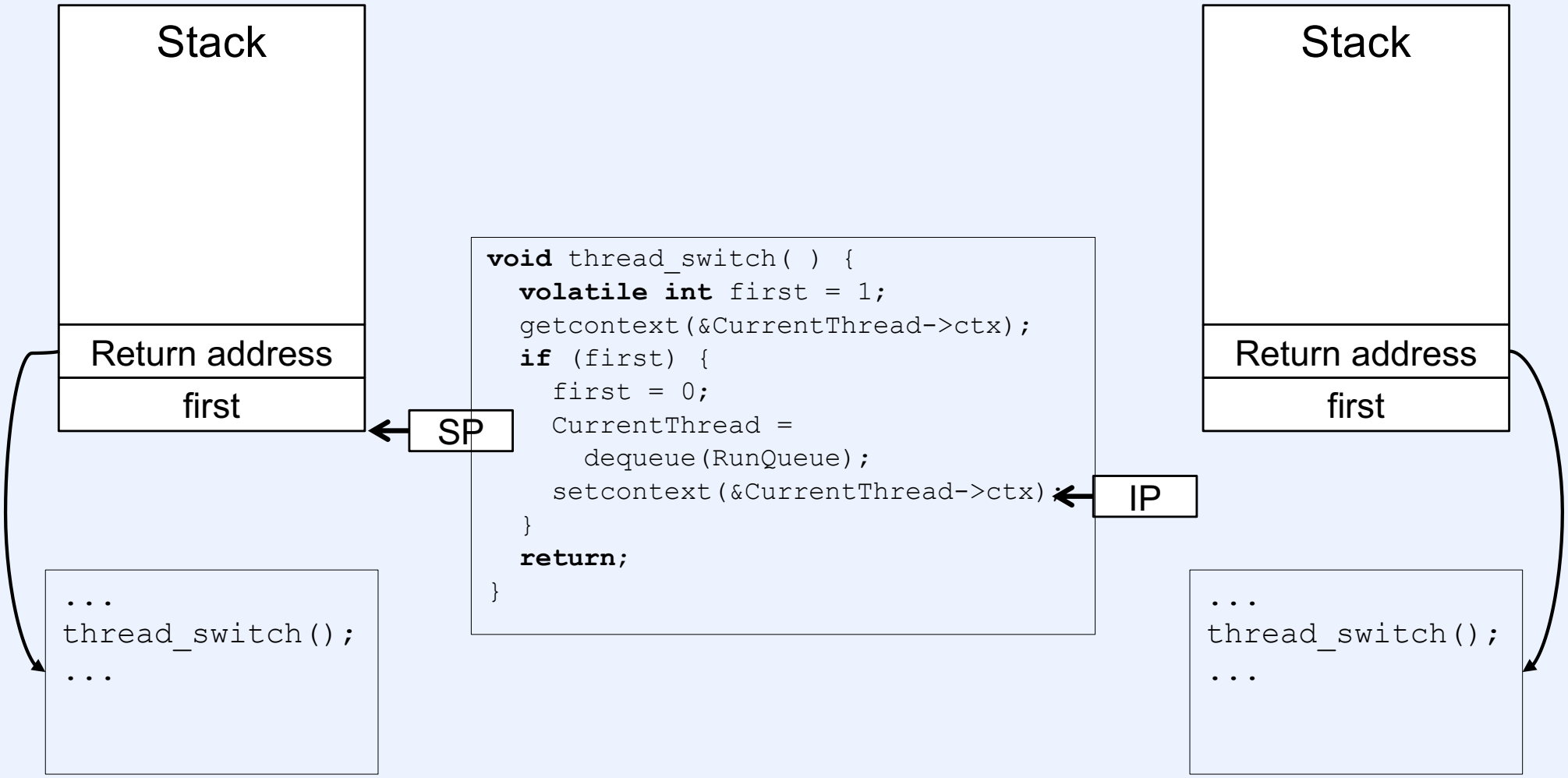
Thread-Switch Exchange



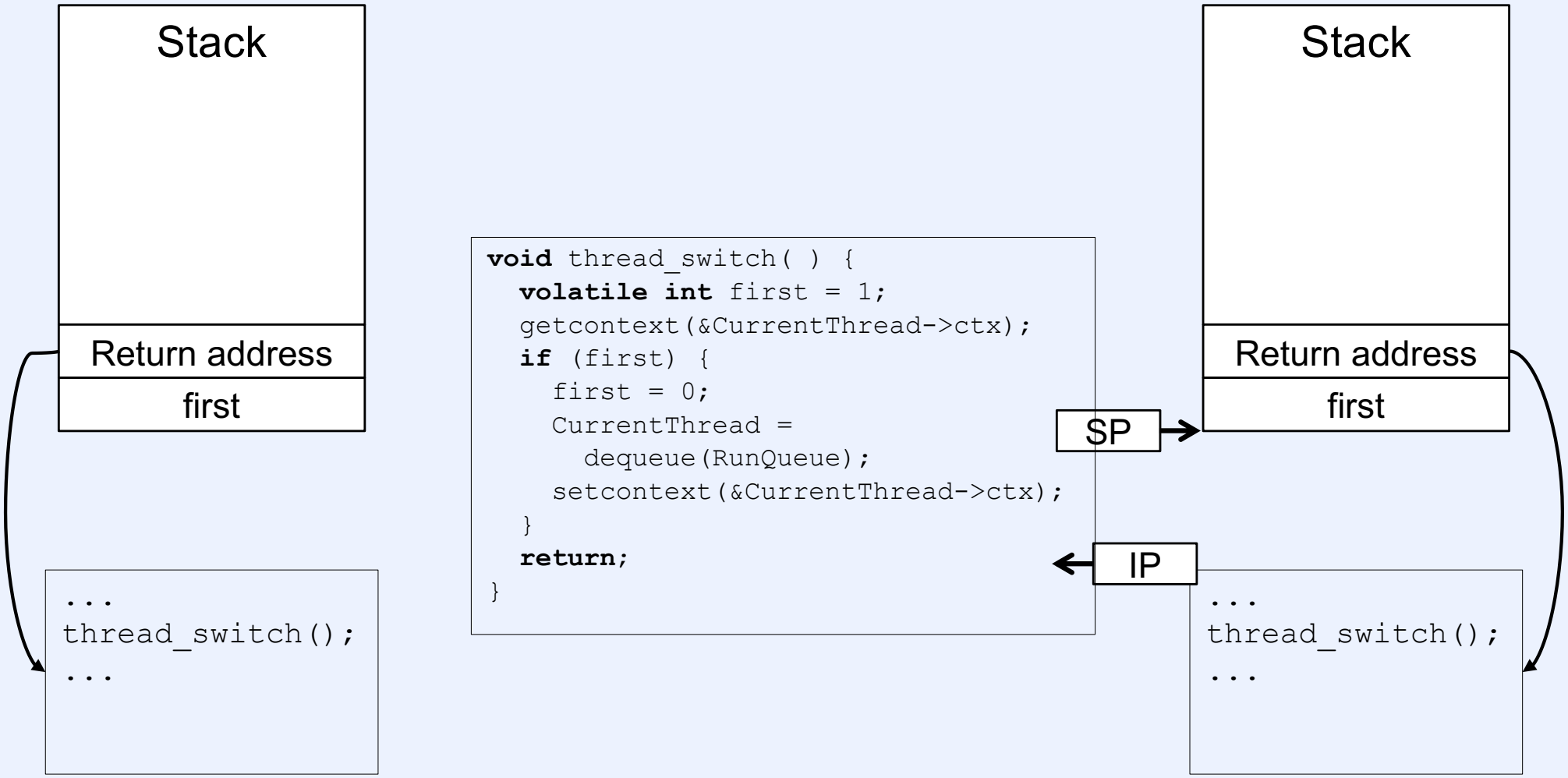
Thread-Switch Exchange



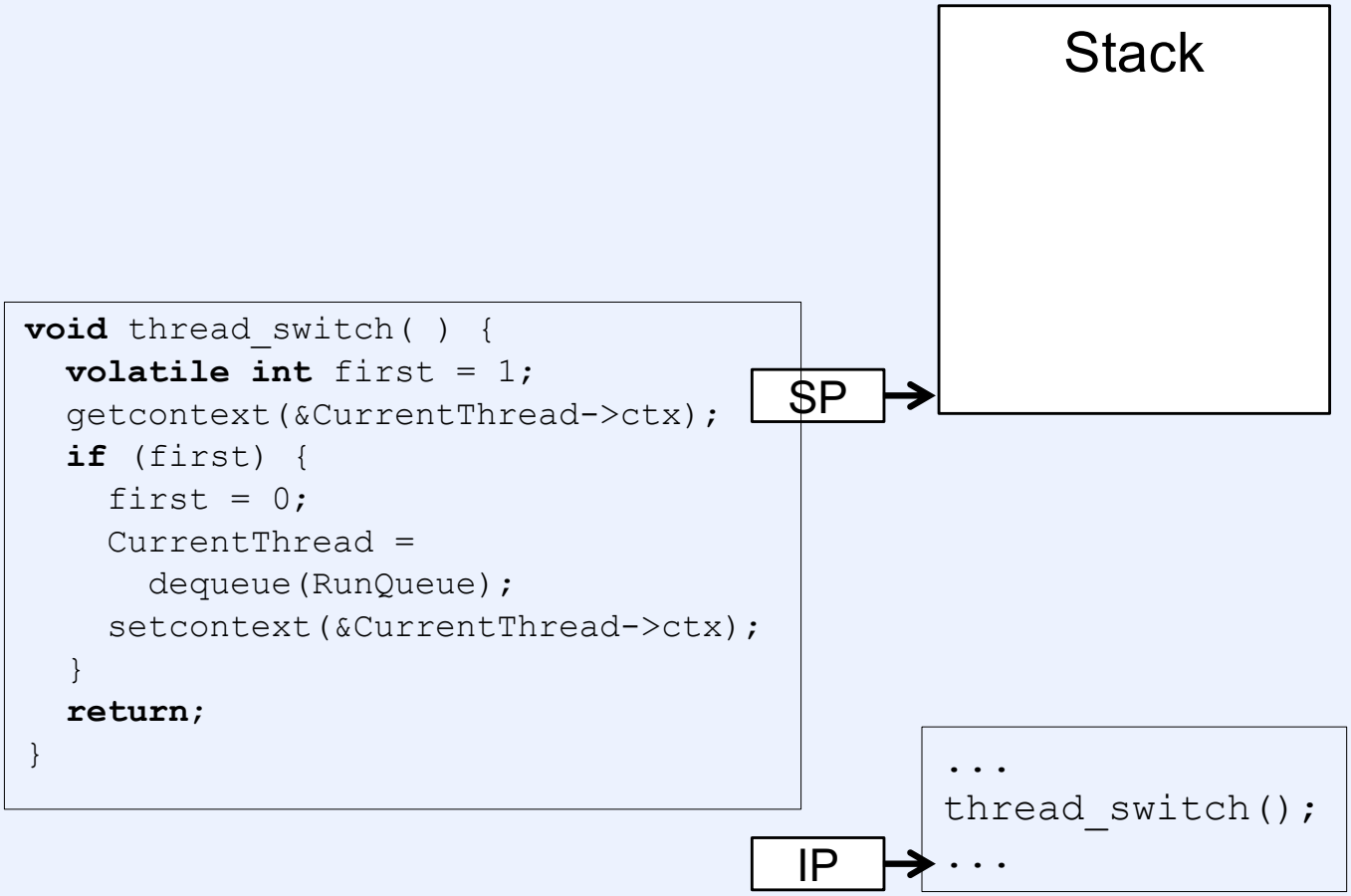
Thread-Switch Exchange



Thread-Switch Exchange



Thread-Switch Exchange



Mutexes

```
mutex_t mut;

mutex_lock(&mut);
x = x+1;
mutex_unlock(&mut);
```

Implementing Mutexes

```
void mutex_lock(mutex_t *m) {
  if (m->locked) {
    enqueue(m->wait_queue, CurrentThread);
    thread_switch();
  }
  m->locked = 1;
}

void mutex_unlock(mutex_t *m) {
  m->locked = 0;
  if (!queue_empty(m->wait_queue))
    enqueue(RunQueue, dequeue(m->wait_queue));
}
```

Quiz 2

```
void mutex_lock(mutex_t *m) {
  if (m->locked) {
    enqueue(m->wait_queue, CurrentThread);
    thread_switch();
  }
  m->locked = 1;
}

void mutex_unlock(mutex_t *m) {
  m->locked = 0;
  if (!queue_empty(m->wait_queue))
    enqueue(RunQueue, dequeue(m->wait_queue));
}
```

a) It works.

b) It works as long as there are no more than two threads.

c) There are situations in which it doesn't work for any number of threads greater than 1.

Implementing Mutexes, Take 2

```
void mutex_lock(mutex_t *m) {
  if (m->locked) {
    enqueue(m->queue, CurrentThread);
    thread_switch();
  } else
    m->locked = 1;
}

void mutex_unlock(mutex_t *m) {
  if (queue_empty(m->queue))
    m->locked = 0;
  else
    enqueue(runqueue, dequeue(m->queue));
}
```

Thread Termination

- Termination
  - thread becomes zombie
  - if joinable
    - notify waiter, if present
  - if detached
    - disappear
      - thread can't do this by itself!

The Reaper Thread

```
while(1) {
   wait_for_terminated_zombie()
   delete(zombie);
}
```

Thread Yield

Thread Yield Details

```
void thread_yield() {
  if (!queue_empty(runqueue)) {
    enqueue(runqueue, CurrentThread);
    thread_switch();
  }
}
```

Time Slicing

- **Periodically**
  - **current thread forced to do a thread yield**

```
void ClockInterrupt(int sig) {
  thread_yield();
}
```

- **Implement ClockInterrupt with VTALRM signal**