Understanding the Solution for Assignment P3.2

yield()

```
<prolog>
        push   %rbp
        mov    %rsp,%rbp
        sub    $0x10,%rsp
int prevThread = _currentThread;
        mov    0x2019a5(%rip),%eax
        mov    %eax,-0x4(%rbp)

        ...

__asm__
        push   %rbp
        push   %rbx
        push   %r12
        push   %r13
        push   %r14
        push   %r15
        mov    %rsp,0x8(%rdx)
        mov    0x8(%rax),%rsp
        pop    %r15
        pop    %r14
        pop    %r13
        pop    %r12
        pop    %rbx
        pop    %rbp
<epilog>
        leaveq
        retq
```


Markers (e.g., **%rsp**) point at the data, **not** after it, as we read towards higher addresses (which are at the top). Reading 8 bytes at **%rsp** will thus give `return address`.

Red Instructions indicate the last (already) executed instructions.

Higher addresses are at the top of the stack (stack grows downwards)!

```
<prolog>
        push   %rbp
        mov    %rsp,%rbp
        sub    $0x10,%rsp
int prevThread = _currentThread;
        mov    0x2019a5(%rip),%eax
        mov    %eax,-0x4(%rbp)

        ...

__asm__
        push   %rbp
        push   %rbx
        push   %r12
        push   %r13
        push   %r14
        push   %r15
        mov    %rsp,0x8(%rdx)
        mov    0x8(%rax),%rsp
        pop    %r15
        pop    %r14
        pop    %r13
        pop    %r12
        pop    %rbx
        pop    %rbp
<epilog>
        leaveq
        retq
```


According to x86_64 ABI, GCC uses a stack alignment of 16 bytes for 64 bit code and thus allocates more space than needed (16 bytes instead of just 4 bytes for `prevThread`)

```
<prolog>
        push   %rbp
        mov    %rsp,%rbp
        sub    $0x10,%rsp
int prevThread = _currentThread;
        mov    0x2019a5(%rip),%eax
        mov    %eax,-0x4(%rbp)

        ...

__asm__
        push   %rbp
        push   %rbx
        push   %r12
        push   %r13
        push   %r14
        push   %r15
        mov    %rsp,0x8(%rdx)
        mov    0x8(%rax),%rsp
        pop    %r15
        pop    %r14
        pop    %r13
        pop    %r12
        pop    %rbx
        pop    %rbp
<epilog>
        leaveq
        retq
```



```
<prolog>
        push   %rbp
        mov    %rsp,%rbp
        sub    $0x10,%rsp
int prevThread = _currentThread;
        mov    0x2019a5(%rip),%eax
        mov    %eax,-0x4(%rbp)

        ...

__asm__
        push   %rbp
        push   %rbx
        push   %r12
        push   %r13
        push   %r14
        push   %r15
        mov    %rsp,0x8(%rdx)
        mov    0x8(%rax),%rsp
        pop    %r15
        pop    %r14
        pop    %r13
        pop    %r12
        pop    %rbx
        pop    %rbp
<epilog>
        leaveq
        retq
```


At this point we switch to the stack we created in `startThread()`. The layout differs (e.g., no local variables) but that is not a problem as long as we can leave `yield()` correctly.
Note: **%rbp** still points to stack of thread 0.

```
<prolog>
        push   %rbp
        mov    %rsp,%rbp
        sub    $0x10,%rsp
int prevThread = _currentThread;
        mov    0x2019a5(%rip),%eax
        mov    %eax,-0x4(%rbp)

        ...

__asm__
        push   %rbp
        push   %rbx
        push   %r12
        push   %r13
        push   %r14
        push   %r15
        mov    %rsp,0x8(%rdx)
        mov    0x8(%rax),%rsp
        pop    %r15
        pop    %r14
        pop    %r13
        pop    %r12
        pop    %rbx
        pop    %rbp
<epilog>
        leaveq
        retq
```


Alternative Stack Layout

You can also reserve six slots, i.e., one slot for every popq. However, it is important that **%rbp** contains a valid address after its first pop. This is due to the `leaveq`, which would otherwise corrupt the stack pointer **%rsp** with the `mov`.


```
leaveq
> mov %rbp,%rsp
> pop %rbp
```



```
leaveq
> mov %rbp,%rsp
> pop %rbp
```



```
<prolog>
        push   %rbp
        mov    %rsp,%rbp
        sub    $0x10,%rsp
int prevThread = _currentThread;
        mov    0x2019a5(%rip),%eax
        mov    %eax,-0x4(%rbp)

        ...

__asm__
        push   %rbp
        push   %rbx
        push   %r12
        push   %r13
        push   %r14
        push   %r15
        mov    %rsp,0x8(%rdx)
        mov    0x8(%rax),%rsp
        pop    %r15
        pop    %r14
        pop    %r13
        pop    %r12
        pop    %rbx
        pop    %rbp
<epilog>
        leaveq
        retq
```


func()

```
<prolog>
        push   %rbp
        mov    %rsp,%rbp

        ...
```

It is ok, that **%rbp** does not have a meaningful value at the end of `yield()` for a new thread. This is because the function that the thread will start in (`func()`) will immediately build a valid stack frame. The same is true for `_parkThread()`.

Now we are within the thread function. Local variables etc. are allocated on the stack

```
<epilog>
        leaveq
        retq
```


Right before ret Instruction in `func()`. Next we will pop address of `_parkThread()` as return address.

Understanding the Stack Smashing Error

> Compiling without `-fno-stack-protector`

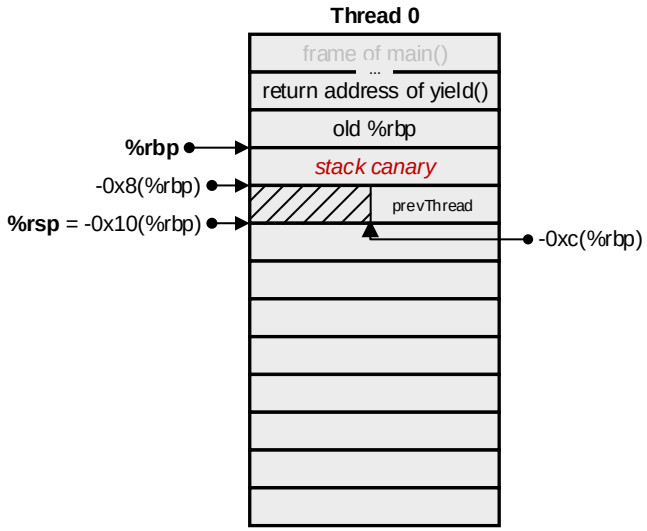
yield()

```
<prolog>
        push   %rbp
        mov    %rsp,%rbp
        sub    $0x10,%rsp

        mov    %fs:0x28,%rax               # Load canary value into %eax
        mov    %rax,-0x8(%rbp)             # Store canary value on the stack
        xor    %eax,%eax

int prevThread = _currentThread;
        mov    0x2018c5(%rip),%eax
        mov    %eax,-0xc(%rbp)
        ...
<epilog>
        mov    -0x8(%rbp),%rax              # Load canary value from stack
        xor    %fs:0x28,%rax                # Compare it with original value
        je     400920 <yield+0xc2>          # When equal (no stack corruption) jump to 400920
        callq  4005a0 <__stack_chk_fail@plt> # Otherwise: Terminate with "stack smashing detected"

400920: leaveq
        retq
```



The compiler adds a magic value (stack canary) at the top of the stack. This value is checked at the end of the function. Since the stack we create in `startThread()` does not have the canary on it, the check fails!