

Implementing Threads on the Intel 64

CS 520

Dept. of Computer Science

Univ. of New Hampshire

Goal

Provide illusion of multiple threads executing when there is only a single processor.

```
long thread_create(void (*work)(void *), void *info);  
void thread_yield(void);
```

Thread State

What is a thread?

PC +
other registers +
stack

Thread Control Block (TCB)

contains thread state when it is
not running

Keep list of TCBs for ready-to-run threads
↳ the ready list

my convention: head of ready list is the
current running thread

thread-yield

rotate TCBs on ready list

First becomes last

second becomes first

save state of running thread
into its TCB

restore state of next thread

from its TCB

changing stacks

when you restore the stack pointer
of the next thread, you change
from running on one stack
to running on a second stack

this is the magic moment

↳ Cool, Cool, Cool!

Complication

we can create a TCB for a thread
when we create it

but we don't create the main thread

So first time a thread primitive is
called, create a TCB for the
main thread and make it the
sole node on the ready list

saving/restoring thread state

need to access registers

so code needs to be in assembly language

```
void asm_yield(TCB *cur, TCB *next);
```

which registers need to be saved?

rsp

rbx, r12 - r15 i.e. callee saved

why not rbp? why not rip?

TCB will also contain slots for rdi & rsi

↳ part of strategy for starting a thread

accessing a struct from assembly language

rdi



rsp	0
rbx	8
rdi	16
rsi	24
r12	32
r13	40
r14	48
r15	56
next	64
	...
	...
	...

movq %r12, 32(%rdi) # save

movq 32(%rsi), %r12 # restore

└ if rsi points to next thread's TCB

i.e. `qsm_yield(curTCB, nextTCB)`
 ↑ ↑
 rdi rsi

The Magic Moment

asm_yield:

```
pushq %rbp  
movq %rsp, %rbp
```

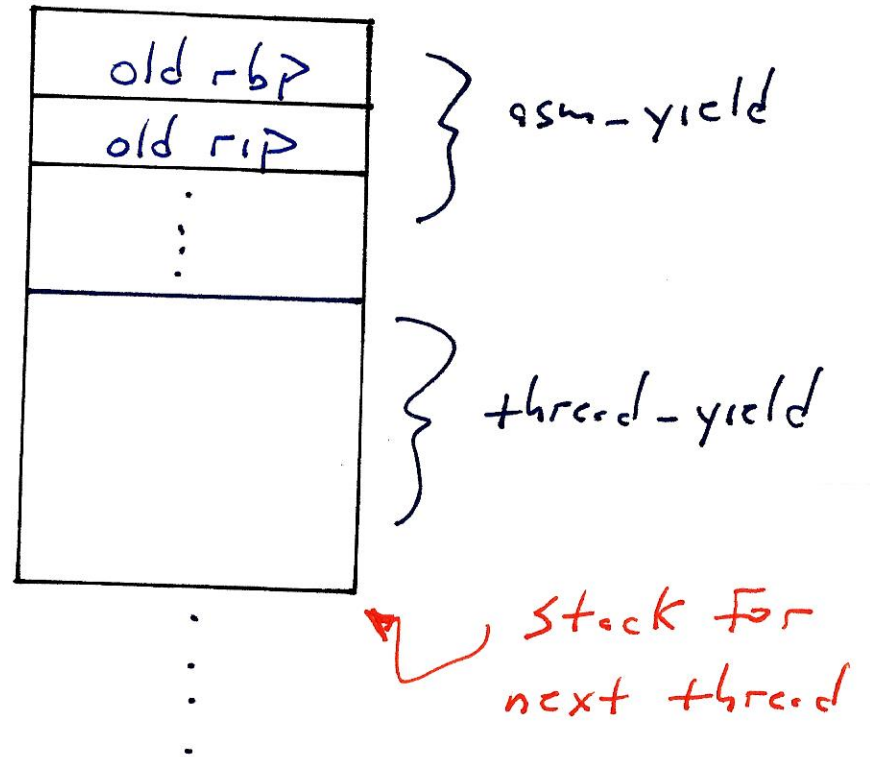
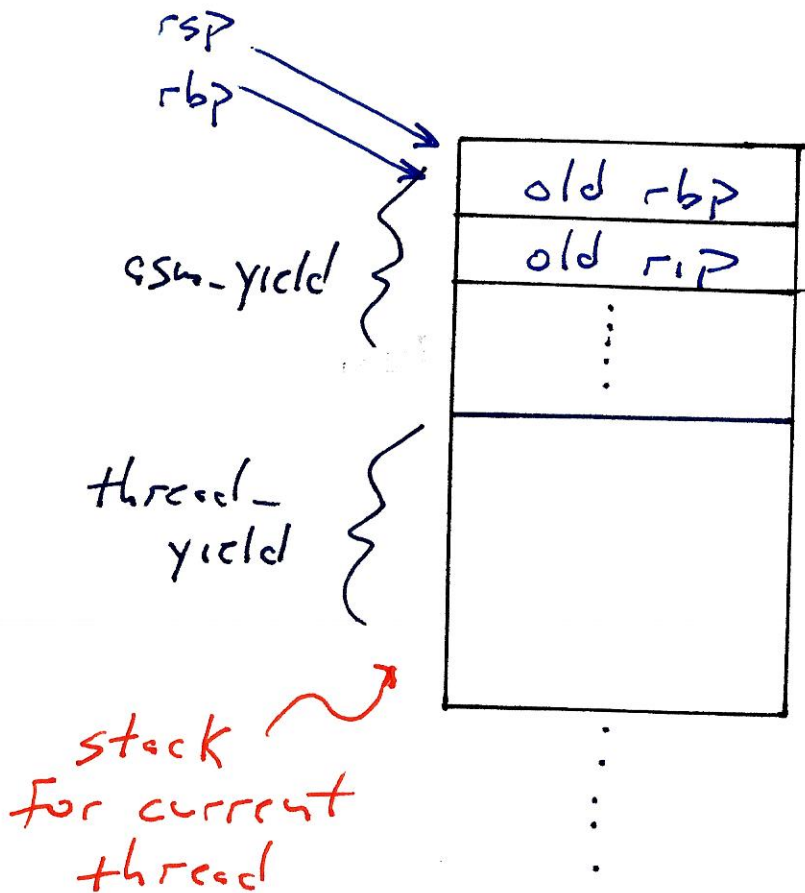
```
...
```

```
movq (%rsi), %rsp
```

```
popq %rbp
```

```
ret
```

Here!



thread_create (work, info)

create TCB and put it on the end
of the ready list

use malloc to allocate a stack

↳ save base of stack in the TCB
so it can be freed later

initialize TCB & stack so that the
thread will execute this function

```
void thread_start (void (*work)(void*), void *info)
{
    work (info);
    thread_cleanup ();
}
```

thread_create return value

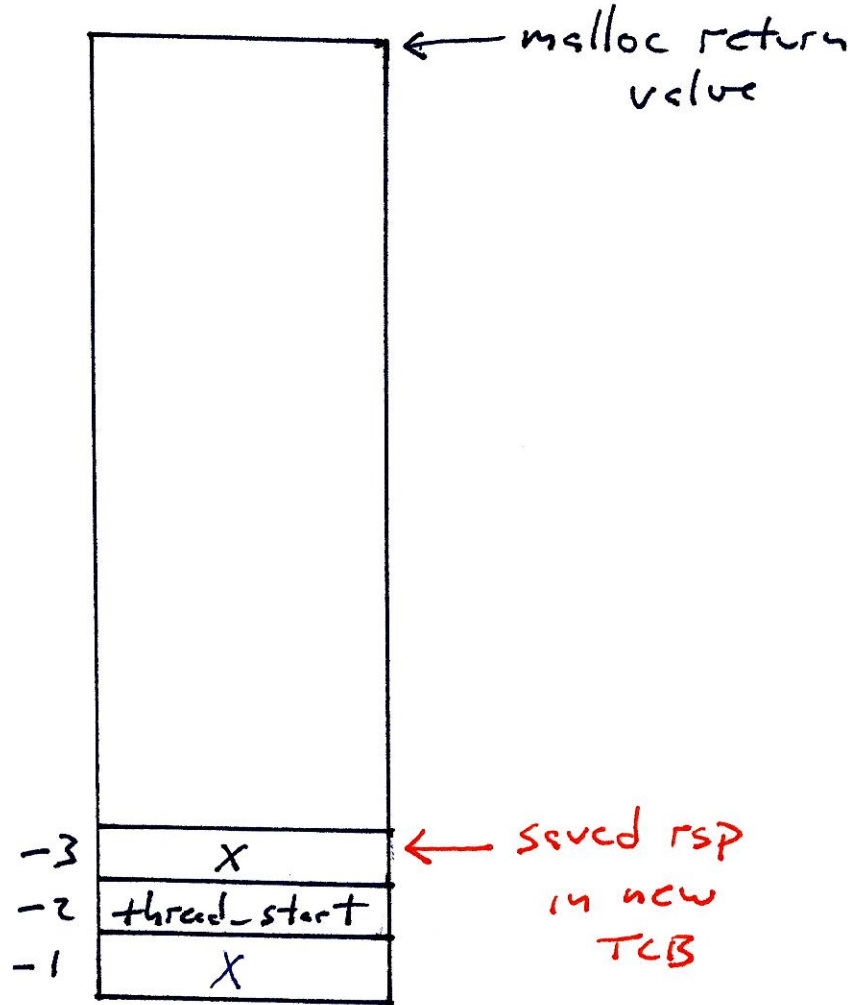
address of TCB

serves as "thread ID"

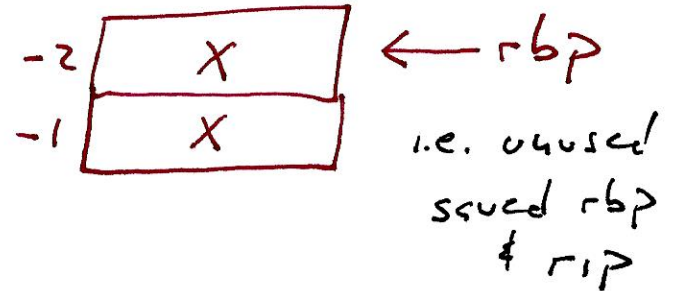
TCB & stack initialization

```
asm_yield:
    :
    popq %rbp
    ret
```

low



```
thread_start:
    pushq %rbp
    movlq %rsp, %rbp
```



high

new stack

* put thread_start's args in rdi & rsi slots of new TCB

thread_cleanup

remove TCB from front of ready list

Free stack*

Free TCB

call asm_yield(NULL, next)

↑
TCB now on front
of ready list

* dangerous to free stack that we are still
executing on.

probably should defer frees of stack by one call

↳ Free this stack the next time thread_cleanup
executes

Wow!

do you believe?

Preemptive Scheduling

"hide" calls to `thread_yield`

ask O.S. to generate timer signals
at regular intervals

install handler function for signal
that will call `thread_yield`

must disable signal handler at
critical points

↳ when creating thread
when thread is being cleaned up
etc