

- ☑ 1. Last time
- ☑ 2. Software architecture: device drivers
- ☑ 3. Synchronous vs. asynchronous I/O
- ☑ 4. User-level threading, intro
- ☑ 5. Context switches (user-level threading)

switch()

yield()

I/O

- ☑ 6. Cooperative multithreading
- ☑ 7. Preemptive user-level multithreading

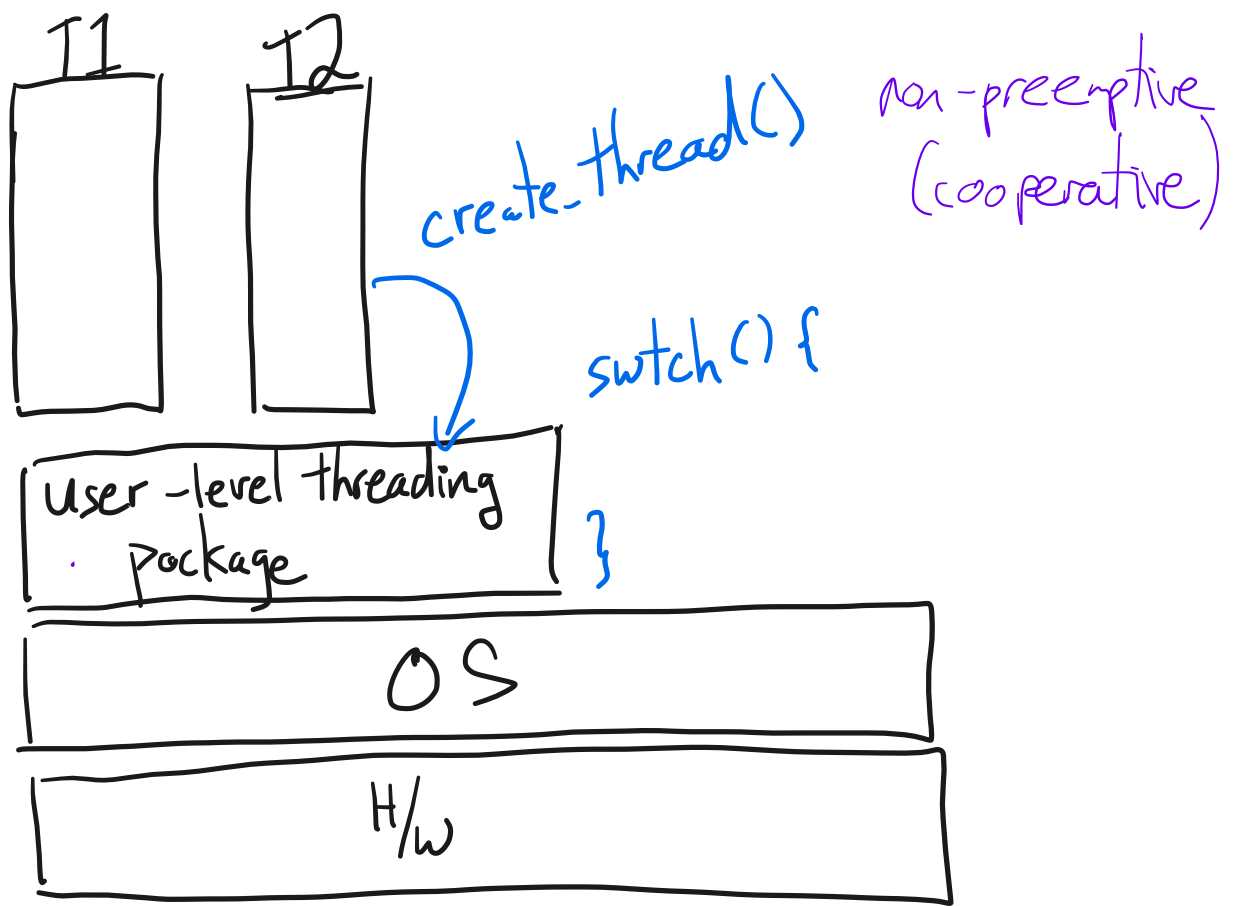
---

2., 3. See whiteboard from prior class

---

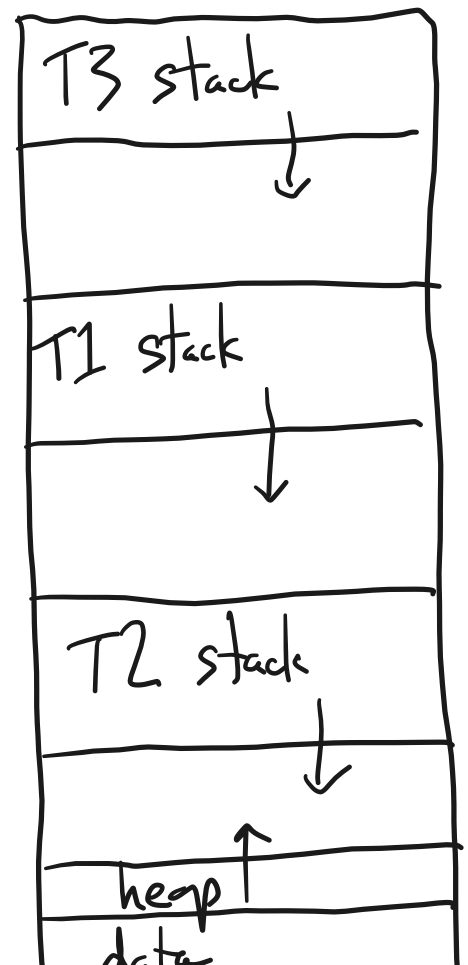
4. User-level threading

preemptive

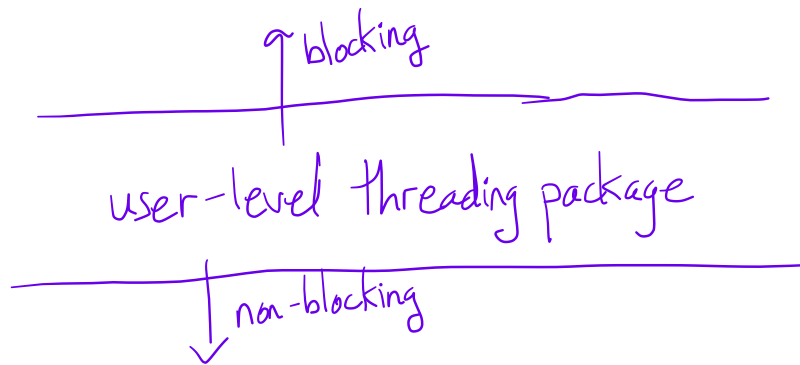


## 5. Context switches (user space)

- switch registers active
- switch page tables



text



Nov 02, 25 9:05

switch.txt

Page 1/2

```

1 CS 202
2 Handout 11 (Class 17)
3
4 1. User-level threads and switch()
5
6 We'll study this in the context of user-level threads.
7
8 Per-thread state in thread control block:
9
10 typedef struct tcb {
11     unsigned long saved_rsp; /* Stack pointer of thread */
12     char *t_stack; /* Bottom of thread's stack */
13     /* ... */
14 };
15
16 Machine-dependent thread initialization function:
17
18 void thread_init(tcb **t, void (*fn) (void *), void *arg);
19
20 Machine-dependent thread-switch function:
21
22 void swtch(tcb *current, tcb *next);
23
24 Implementation of swtch(current, next):
25
26 # gcc x86-64 calling convention:
27 # on entering switch():
28 # register %rdi holds first argument to the function ("current")
29 # register %rsi holds second argument to the function ("next")
30
31 # Save call-preserved (aka "callee-saved") regs of 'current'
32 pushq %rbp
33 pushq %rbx
34 pushq %r12
35 pushq %r13
36 pushq %r14
37 pushq %r15
38
39 # store old stack pointer, for when we swtch() back to "current" later
40 movq %rsp, (%rdi) /* %rdi->saved_rsp = %rsp */
41 movq (%rsi), %rsp /* %rsi->saved_rsp = %rsp */
42
43 # Restore call-preserved (aka "callee-saved") regs of 'next'
44 popq %r15
45 popq %r14
46 popq %r13
47 popq %r12
48 popq %rbx
49 popq %rbp
50
51 # Resume execution, from where "next" was when it last entered switch()
52 ret
53
54

```

Sunday November 02, 2025

%.rsp

T1 (current)

T2 (next)

switch.txt

Nov 02, 25 9:05

switch.txt

Page 2/2

```

55
56 2. Example use of switch(): the yield() call.
57
58 A thread is going about its business and decides that it's executed for
59 long enough. So it calls yield(). Conceptually, the overall system needs
60 to now choose another thread, and run it:
61
62 void yield() {
63
64     tcb* next = pick_next_thread(); /* get a runnable thread */
65     tcb* current = get_current_thread();
66
67     swtch(current, next);
68     /* when 'current' is later rescheduled, it starts from here */
69 }
70
71 3. How do context switches interact with I/O calls?
72
73 This assumes a user-level threading package.
74
75 The thread calls something like "fake_blocking_read()". This looks
76 to the _thread_ as though the call blocks, but in reality, the call
77 is not blocking:
78
79 int fake_blocking_read(int fd, char* buf, int num) {
80
81     int nread = -1;
82
83     while (nread == -1) {
84
85         /* this is a non-blocking read() syscall */
86         nread = read(fd, buf, num);
87
88         if (nread == -1 && errno == EAGAIN) {
89             /*
90              * read would block. so let another thread run
91              * and try again later (next time through the
92              * loop).
93              */
94             yield();
95         }
96     }
97
98     return nread;
99 }
100
101
102
103
104
105

```

f1()

yield()  
swtch();

f2()

yield()  
swtch();

0xf...f500

1/1



~~0xf...f100~~  
0xf...f500

saved\_rsp

t\_stack

|            |            |
|------------|------------|
| 0xf...f100 | 0xf...f500 |
| 0xf...f200 | 0xf...f600 |