☑ 1. Last time

☑ 2. Context switches (WeensyOS)

ONE HANDOUT

☑ 3. User-level threading, intro

☑ 4. Context switches (user-level threading)

- swtch()
- yield()
- I/o

☑ 5. Cooperative multithreading

□ 6. Preemptive user-level multithreading

☑ 7. mmap() again

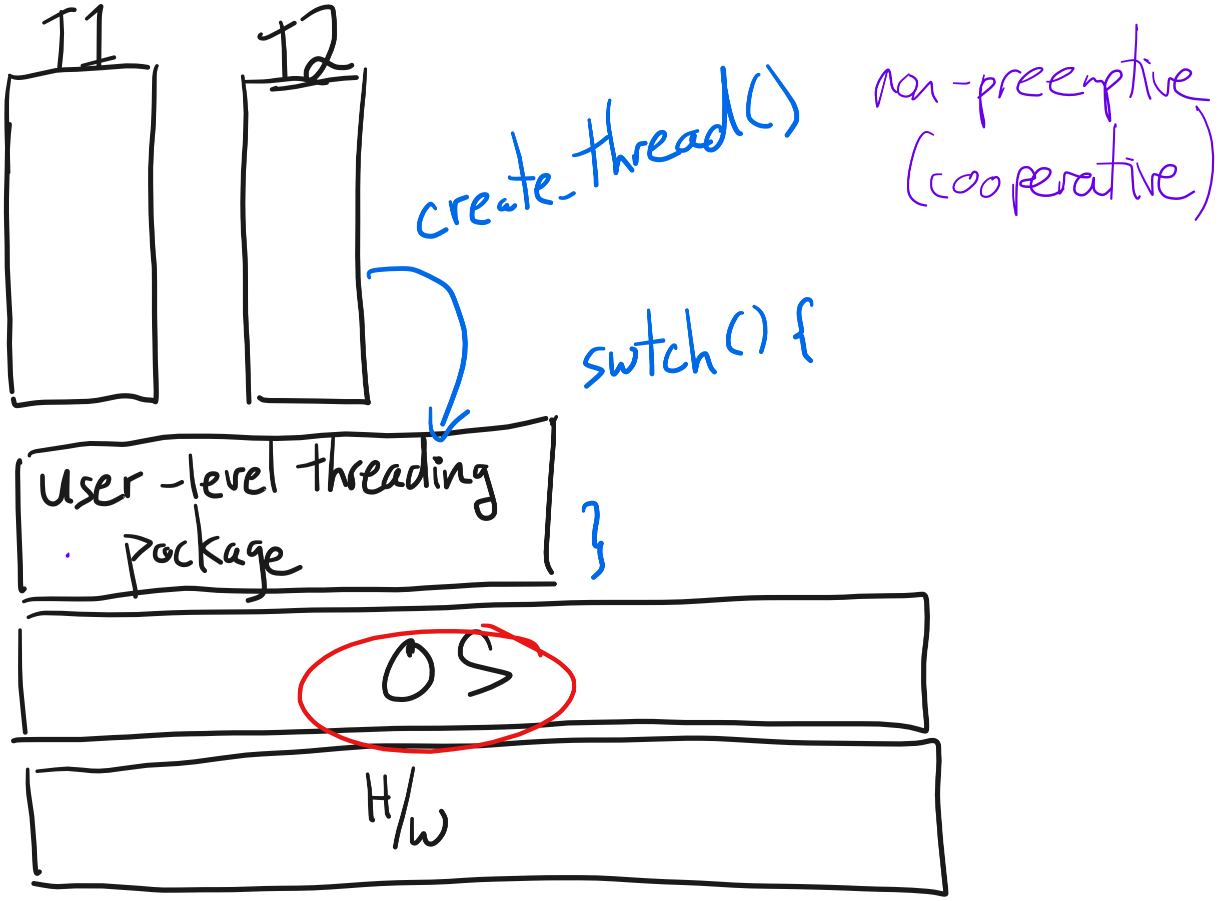
	k	u
pre	✓	✓
non-pre coop	✗	✓*

2. Context switches in WeensyOS

(see pictures at the end)

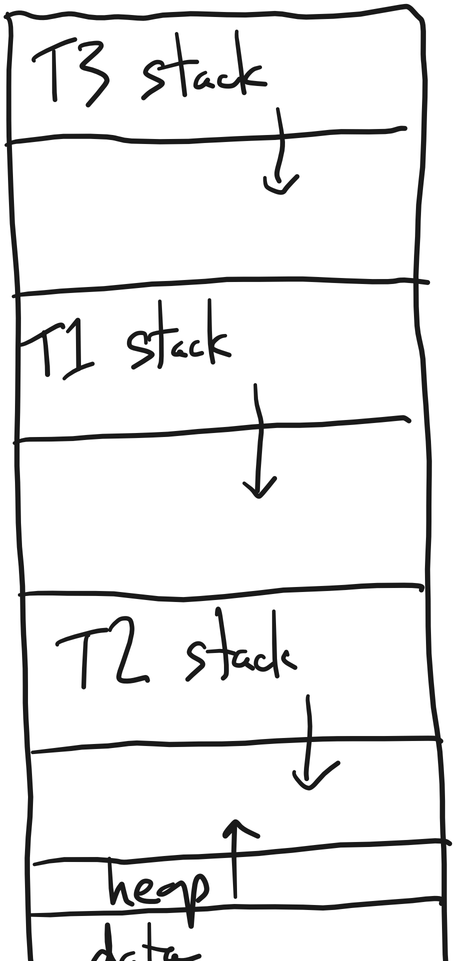
3. User-level threading

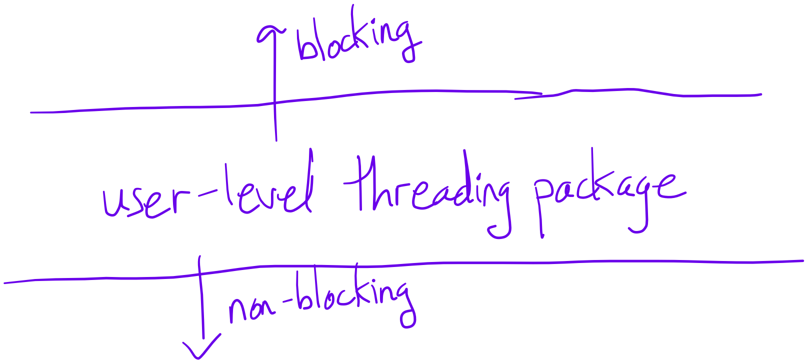
preemptive



4. Context switches (user space)

- switch registers active
- ~~switch page tables~~



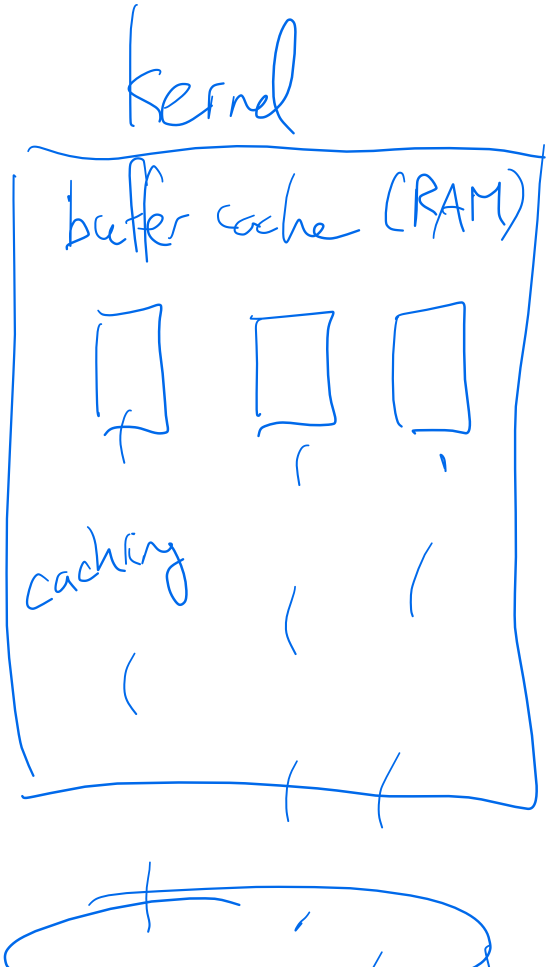


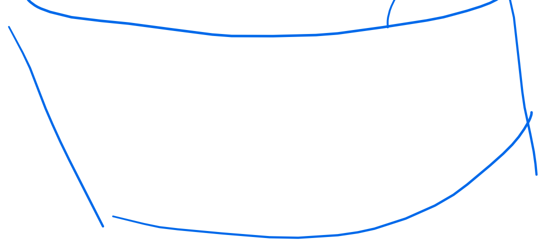
mmap, continued

mmap

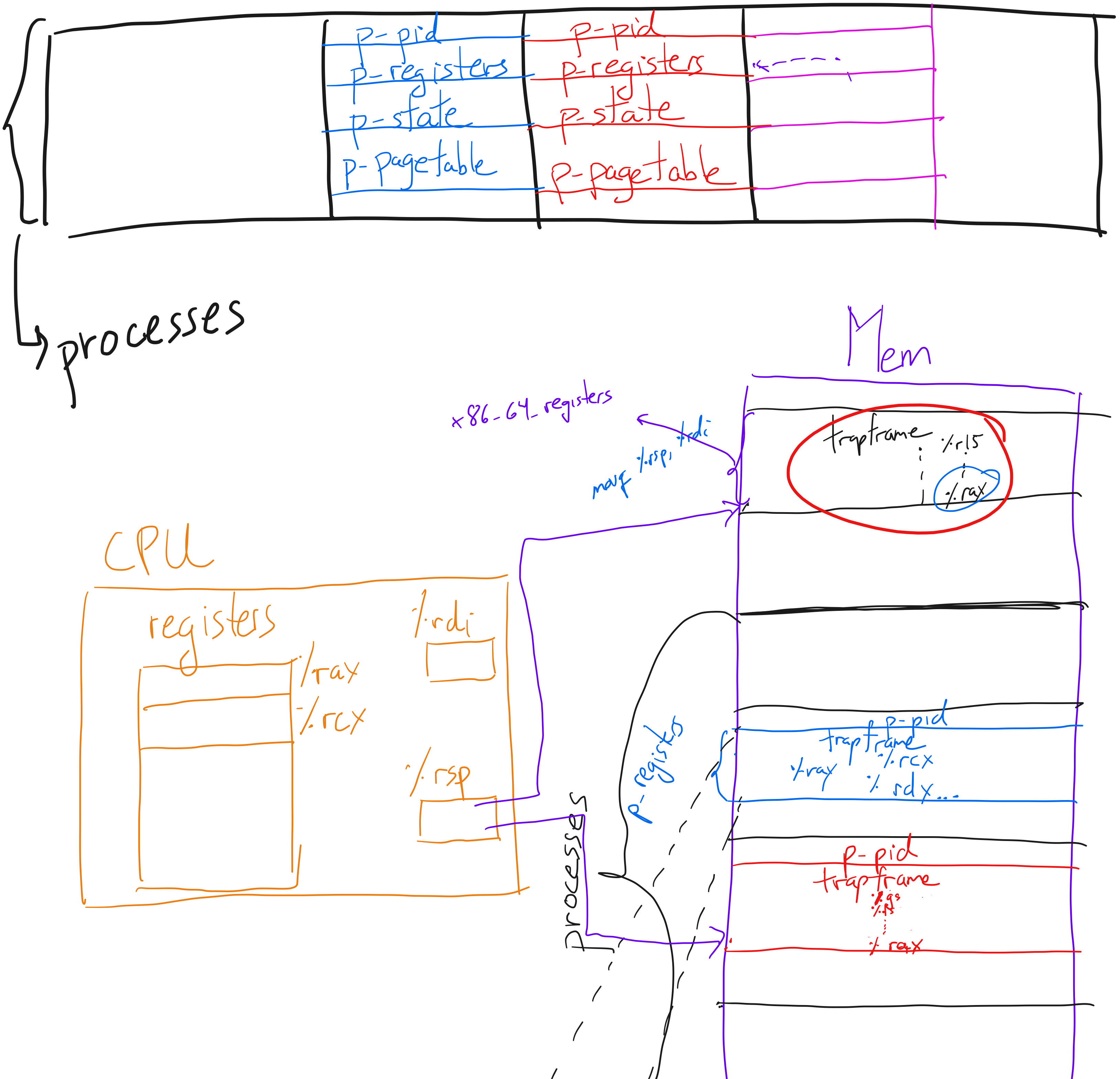
PT:

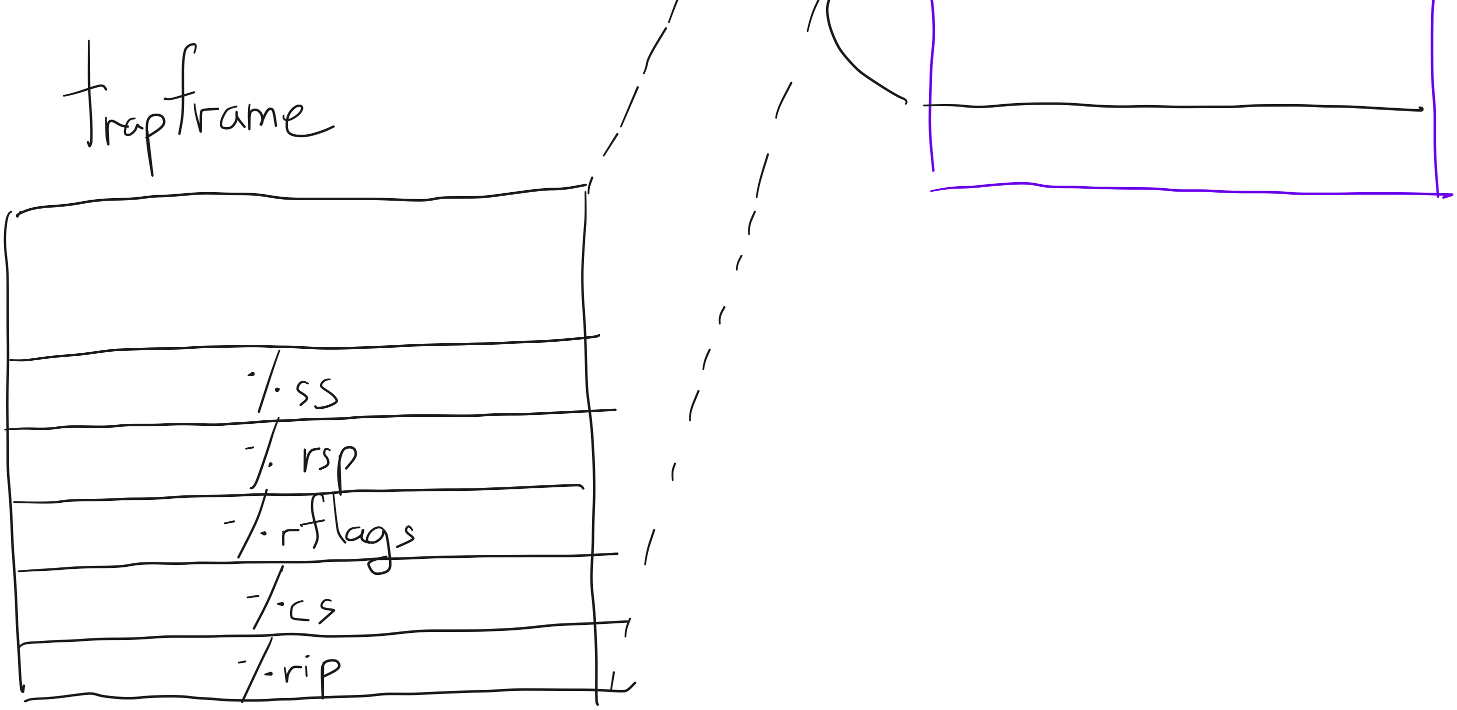
VA of "file" → phys addr in RAM (in buffer cache)



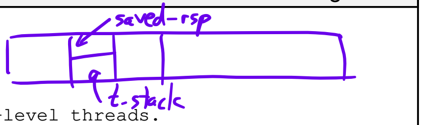


Context switches in Weensy OS




trapframe
%ss
%rsp
%rflags
%cs
%rip

CS 202, Spring 202~~3~~4
Handout 10 (Class 17)



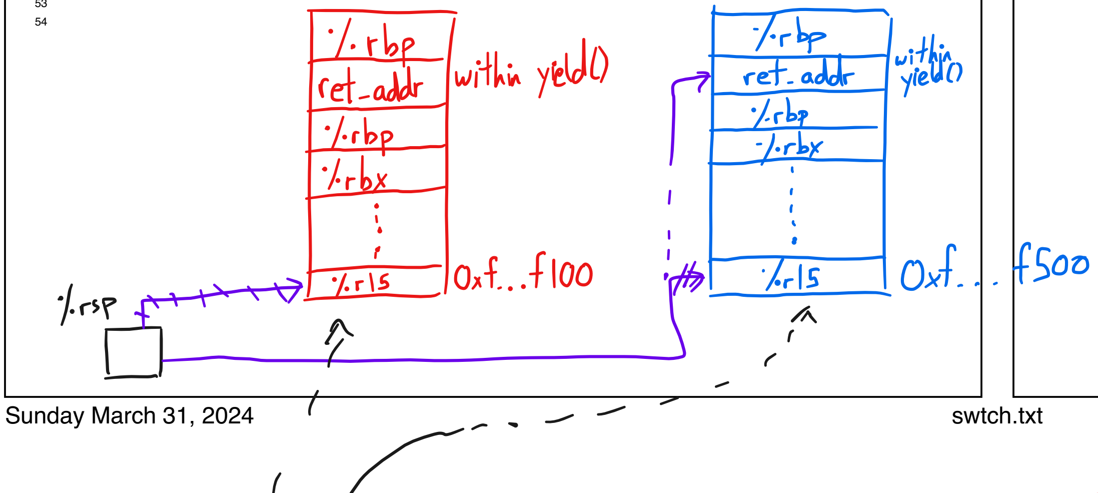
```
1. User-level threads and swtch()

    We'll study this in the context of user-level threads.

    Per-thread state in thread control block:

        typedef struct tcb {
            unsigned long saved_rsp;     /* Stack pointer of thread */
            char *t_stack;               /* Bottom of thread's stack */
            /* ... */
        };
```

↑* "bottom" means higher mem address because stack grows down.

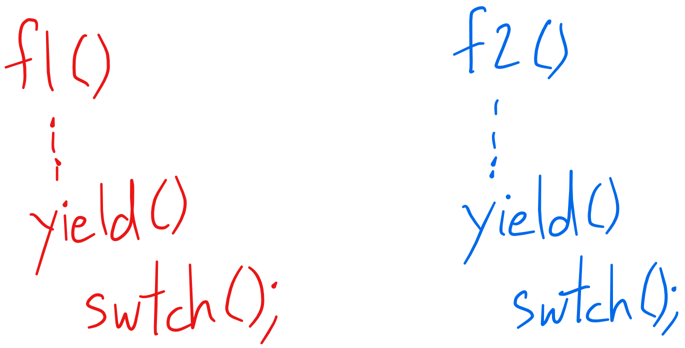
```
    Machine-dependent thread initialization function:

        void thread_init(tcb **t, void (*fn) (void *), void *arg);

    Machine-dependent thread-switch function:

        void swtch(tcb *current, tcb *next);
```

red (current) blue (next)

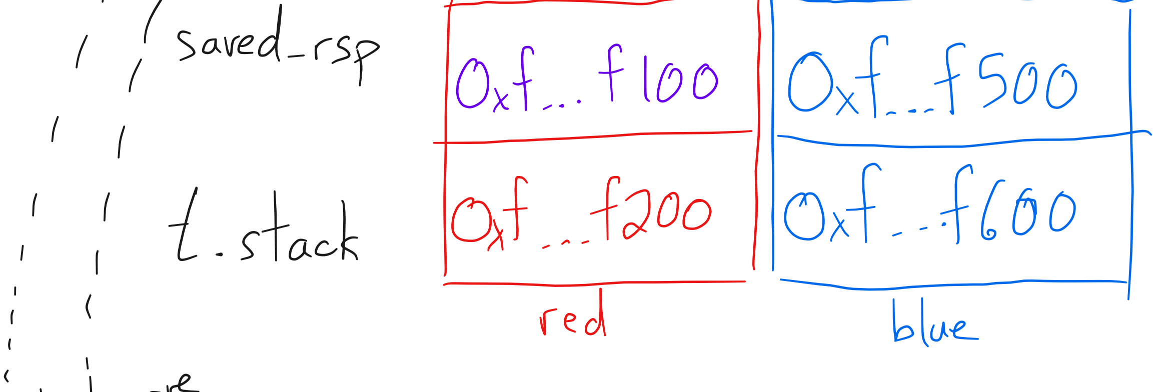
```
    Implementation of swtch(current, next):

        # gcc x86-64 calling convention:
        # on entering swtch():
        #  register %rdi holds first argument to the function ("current")
        #  register %rsi holds second argument to the function ("next")

        # Save call-preserved (aka "callee-saved") regs of 'current'
        pushq %rbp
        pushq %rbx
        pushq %r12
        pushq %r13
        pushq %r14
        pushq %r15

        # store old stack pointer, for when we swtch() back to "current" later
        movq %rsp, (%rdi)                       # %rdi->saved_rsp = %rsp
        movq (%rsi), %rsp                       # %rsp = %rsi->saved_rsp

        # Restore call-preserved (aka "callee-saved") regs of 'next'
        popq %r15
        popq %r14
        popq %r13
        popq %r12
        popq %rbx
        popq %rbp

        # Resume execution, from where "next" was when it last entered swtch()
        ret
```



```
2. Example use of swtch(): the yield() call.

    A thread is going about its business and decides that it's executed for
    long enough. So it calls yield(). Conceptually, the overall system needs
    to now choose another thread, and run it:

    void yield() {

        tcb* next    = pick_next_thread();   /* get a runnable thread */
        tcb* current = get_current_thread();

        swtch(current, next);

        /* when 'current' is later rescheduled, it starts from here */
    }

3. How do context switches interact with I/O calls?

    This assumes a user-level threading package.

    The thread calls something like "fake_blocking_read()". This looks
    to the _thread_ as though the call blocks, but in reality, the call
    is not blocking:

    int fake_blocking_read(int fd, char* buf, int num) {

        int nread = -1;

        while (nread == -1) {

            /* this is a non-blocking read() syscall */
            nread = read(fd, buf, num);

            if (nread == -1 && errno == EAGAIN) {
                /*
                 * read would block. so let another thread run
                 * and try again later (next time through the
                 * loop).
                 */
                yield();
            }
        }

        return nread;
    }
```



TCBs



	red	blue
saved_rsp	0xf...f100	0xf...f500
t.stack	0xf...f200	0xf...f600

these stacks are inside the single address space. See accompanying diagram "Context switches (user space)" earlier in these notes.