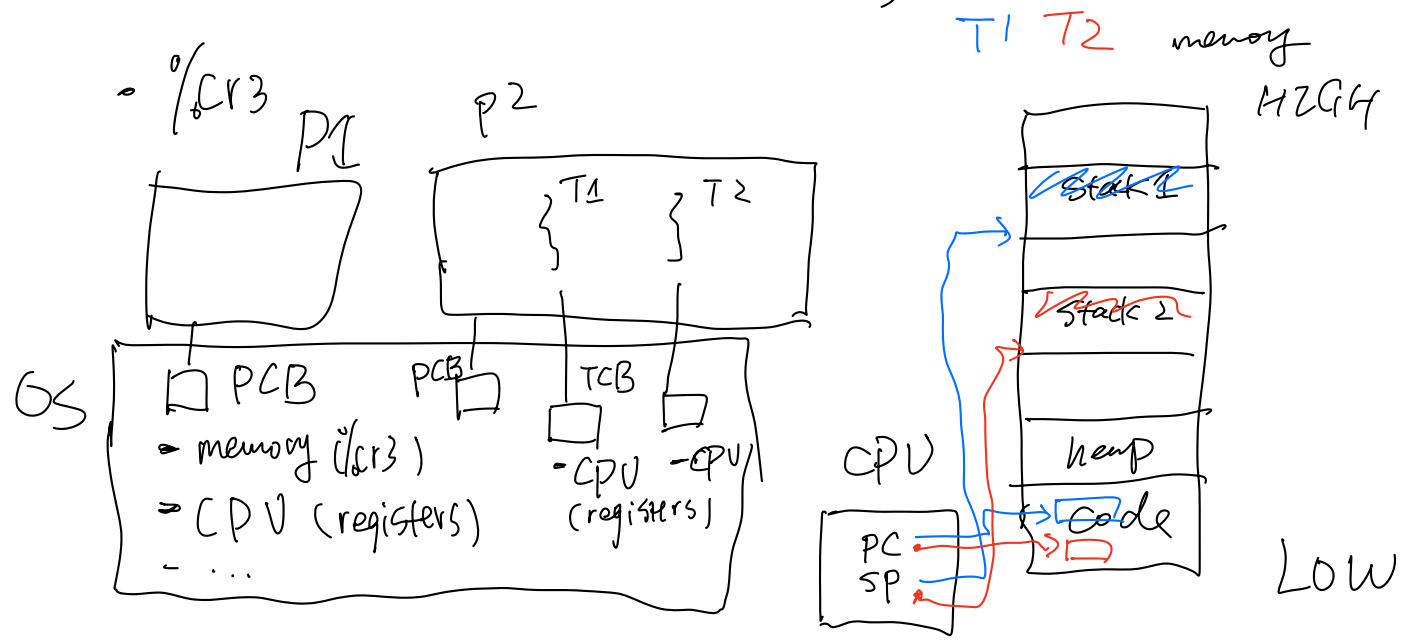
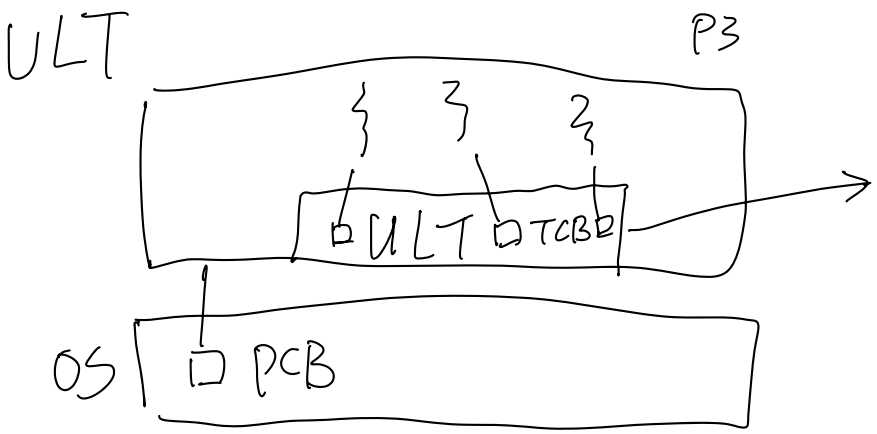
Week 3.a
CS6640
09/19 2023
https://naizhengtan.github.io/23fall/

1. Review threading
2. Context switch in user-space
3. Cooperative vs. Preemptive multithreading in user-level

Q: Processes vs. threads. (kernel-level)

ULT





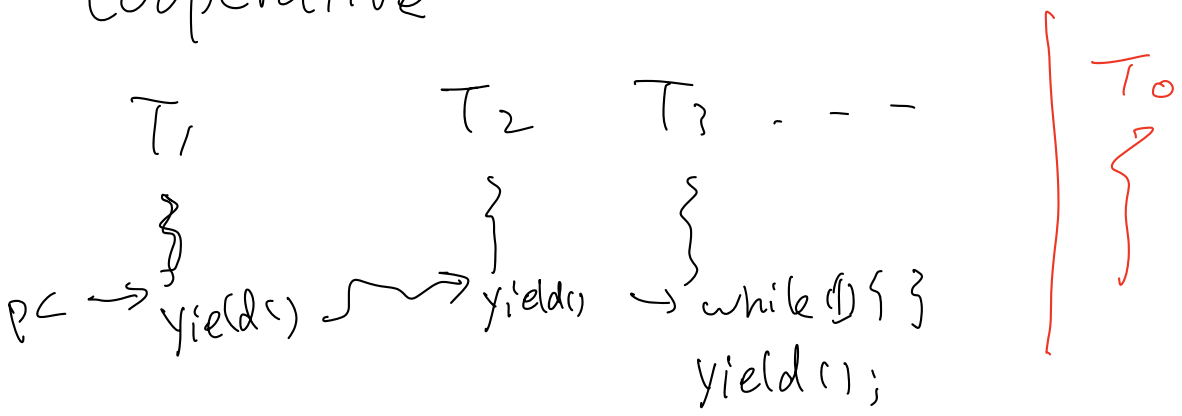
responsibility:
1. manage TCBs
2. Create stack for new threads
3. Scheduling
4. Context switch!

Q: heap for new threads?

① ② ③ ④
stack, heap, data, code

① + ④ ⇐
④ + ③ ⇐

- CtX switching
  - ~~① memory view~~
  - ② registers CPU → PC, SP, other registers.
- Cooperative



- preemptive ULT

  signal $kill -9 PID

CS6640 Handout Week2.b

1. Background: RISC-V assembly I

a) registers
[see "RISC-V registers" in reference page]

b) **addi** rd, rs1, immediate

rd = rs1 + immediate

c) **sw** rs2, offset(rs1)

Memory[rs1 + offset] = rs2

d) **mv** rd, rs1

rd = rs1

e) **call** rd, symbol

rd = pc+4
pc = &symbol

If rd is omitted, ra is implied.

f) **ret**

pc = ra

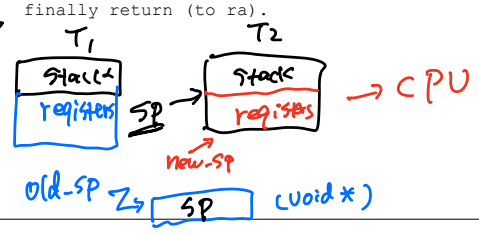
2. Context switch in user-space:

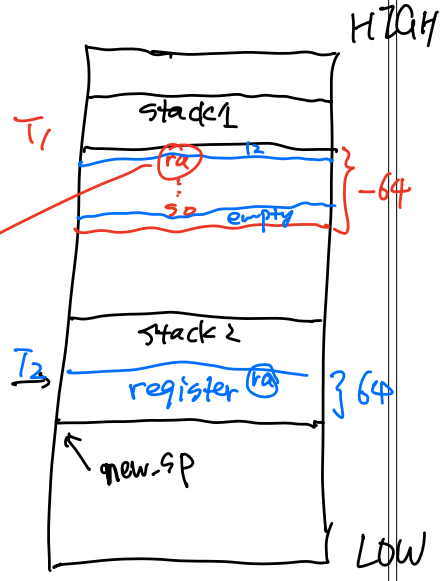
a) void ctx_start(void** old_sp, void* new_sp);

This will be used when starting a new thread.
It will save registers on the old stack,
store current stack pointer to "old_sp",
switch stack to the "new_sp",
and
finally call ctx_entry().

b) void ctx_switch(void** old_sp, void* new_sp);

This will be used for context switch.
It will save registers on the old stack,
store current stack pointer to "old_sp",
switch stack to the "new_sp",
and
restore registers from the new stack,
finally return (to ra).





3. grass/context.S

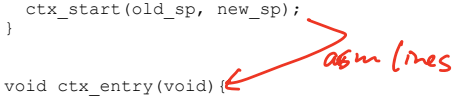
```
 1  ctx_start:
 2      addi sp,sp,-64
 3      sw s0,4(sp)        /* Save callee-saved registers */
 4      sw s1,8(sp)
 5      sw s2,12(sp)
 6      sw s3,16(sp)
 7      sw s4,20(sp)
 8      sw s5,24(sp)
 9      sw s6,28(sp)
10      sw s7,32(sp)
11      sw s8,36(sp)
12      sw s9,40(sp)
13      sw s10,44(sp)
14      sw s11,48(sp)
15      sw ra,52(sp)       /* Save return address */
16      sw sp,0(a0)        /* Save the current stack pointer */
17      mv sp,a1           /* Switch the stack */
18      call ctx_entry     /* Call ctx_entry() */
19
20  ctx_switch:
21      addi sp,sp,-64
22      sw s0,4(sp)        /* Save callee-saved registers */
23      sw s1,8(sp)
24      sw s2,12(sp)
25      sw s3,16(sp)
26      sw s4,20(sp)
27      sw s5,24(sp)
28      sw s6,28(sp)
29      sw s7,32(sp)
30      sw s8,36(sp)
31      sw s9,40(sp)
32      sw s10,44(sp)
33      sw s11,48(sp)
34      sw ra,52(sp)       /* Save return address */
35      sw sp,0(a0)        /* Save the current stack pointer */
36      mv sp,a1           /* Switch the stack */
37      lw s0,4(sp)        /* Restore callee-saved registers */
38      lw s1,8(sp)
39      lw s2,12(sp)
40      lw s3,16(sp)
41      lw s4,20(sp)
42      lw s5,24(sp)
43      lw s6,28(sp)
44      lw s7,32(sp)
45      lw s8,36(sp)
46      lw s9,40(sp)
47      lw s10,44(sp)
48      lw s11,48(sp)
49      lw ra,52(sp)       /* Restore return address */
50      addi sp,sp,64
51      ret
```

Registers

Register	ABI Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP args/return values	Caller
f12-17	fa2-7	FP args	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

4. An example use of ctx_start+ctx_entry

```
void thread_create(void (*f)(void *), void *arg, unsigned int stack_size) {

  tcb = create_thread_control_block();
  old_tcb = current_running_thread_control_block();

  … // do something necessary

  void **old_sp = …   // old stack pointer's address in old_tcb
  void *new_sp =  …   // new stack pointer in tcb

  ctx_start(old_sp, new_sp);
}


void ctx_entry(void){

    … // do something useful

    (*f)(arg);  // run function "f" received by "thread_create"

    … // wrap up
}
```



5. How do context switches interact with I/O?

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;
}
```