Handling User Pointers, Concurrency

```
  why do we need to take special care for user -> kernel?
    security/isolation
    only kernel can touch devices, MMU, FS, other process' state, &c
    think of user program as a potential malicious adversary
```

System call dispatch, arguments and return value

`trap()` calls syscall(), since trapno in this case is T_SYSCALL (0x40).

syscall() dispatches to a function it finds by indexing into the syscalls array. It uses the eax from the trap frame as the index. What is in that eax? Where was it set?

Now we're in sys_open(). Where are the arguments the user program originally passed to `open()`? How can the kernel get at them?

sys_open() calls argint() to get its 2nd argument. Argint calculates the value `cp->tf_esp + 4 + 4*n`. What is this? Why the first 4? Why the `4*n`?

fetchint() checks that the address is not beyond the end of user memory. But addr was just calculated by kernel code (in argint()); since the kernel code is trustworthy, is this check really neccessary?

Why do we do seemingly redundant checks for addr and then addr+4? Can't we just check addr+4?

Why does fetchint() add `p->mem` to addr?

Back to sys_open(). It does its job (which we will talk about later) and finally returns a file descriptor using the ordinary C return statement. syscall() puts that return value in `cp->tf->eax`. Why?

Trap Return

In Unix, traps often get translated into signals to the process. Some traps, though, are (partially) handled internally by the kernel -- which ones?

Device Interrupts

trap(), when it's called for a time interrupt, does just two things: increment the ticks variable, and call wakeup. At the end of trap, xv6 calls yield. as we will see, may cause the interrupt to return in a different process!

Locking

Why locks?

The point of a multiprocessor is to increase total performance by running different tasks concurrently on different CPUs, but concurrency can cause incorrect results if multiple tasks try to use the same variables at the same time. Coordination techniques prevent concurrency in situations where the programmer has decided that serial execution is required for correctness.

Diagram: CPUs, bus, memory.

ide.c has a linked list.

List and insert example:

```
struct List {
  int data;
  struct List *next;
};

List *list = 0;

insert(int data) {
  List *l = new List;
  l->data = data;
  l->next = list;  // A
  list = l;        // B
}
```

Whoever wrote this code probably believed it was correct, in the sense that if the list starts out correct, a call to insert() will yield a new list that has the old elements plus the new one. And if you call insert(), and then insert() again, the list should have two new elements. Most programmers write

code that's only correct under **serial** execution - in this case, insert() is only correct if there is only one insert() executing at a time.

What could go wrong with concurrent calls to insert(), as might happen on a multiprocessor? Suppose two different processors both call `insert()` at the same time. If the two processors execute A and B interleaved, then we end up with an incorrect list. To see that this is the case, draw out the list after the sequence A1 (statement A executed by processor 1), A2 (statement A executed by processor 2), B2, and B1.

This kind of error is called a race (the name probably originates in circuit design). The problem with races is that they are difficult to reproduce. For example, if you put print statements in to debug the incorrect behavior, you might change the timing and the race might not happen anymore.

What's required for insert() to be correct if executed on multiple CPUs? The overall property we want is that both new elements end up appearing in the list. One way to achieve that is to somehow ensure that only one processor at a time is allowed to be executing inside `insert()`; let one of the simultaneous calls proceed, and force the other one to wait until the first has finished. Then two simultaneous executions will have the same result as two serial executions. Assuming `insert()` was correct when executed serially, it will then be correct on a multiprocessor too.

This happens whenever there are two independent computations on *shared state*. e.g., you edit `foo.c` using `vi` and compile it using `gcc`. As a matter of habit we save the source file and wait for the confirmation message before starting the compile job. If we did not wait for the save to finish and started compile simultaneously, what could happen?

Concurrency in real world: one computer, many jobs; one road, many cars; one classroom, many classes. Different solutions for each.

Another example: lost update.

Increment number of hits on a website in each thread. `hits` is a shared variable.

```
hits++;
```

This C statement may get compiled to multiple assemble statements:

```
ld  register, [hits]
add register, register, 1
st  register, [hits]
```

If two threads simultaneously execute this code, what are the possible values of `hits`?

More race condition fun:

```
Thread a:
i = 0;
while (i < 10)
    i = i + 1;
print "A won!";

Thread b:
i = 0;
while (i > -10)
    i = i - 1;
print "B won!";
```

Who wins? Guaranteed someone wins?

Possible solutions:

- Do nothing: In some cases, this may be acceptable. For example, a website may not care if some updates to `hits` are lost, because this event is likely to be rare. But the same cannot be said for other examples. Even rare races can be menacing.
- Don't share, but duplicate state: Sometimes possible. For example, each thread could maintain a private `hits[i]` counter which can be summed up at the end of the day
- General solution? Prevent *bad* interleavings

The lock abstraction

We'd like a general tool to help programmers force serialization. There are a number of things we'd like to be able to express for lists:

1. It's not enough to serialize just calls to insert(); if there's also delete(), we want to prevent a delete() from running at the same time as an insert().
2. Serializing *all* calls to insert() is too much: it's enough to serialize just calls that refer to the same list.
3. We don't need to serialize all of insert(), just lines A and B.

We're looking for an abstraction that gives the programmer this kind of control. The reason to allow concurrency when possible (items 2 and 3) is to increase performance.