Overview (14/10)

By default, newly created threads do not have FPU access. It must be explicitly granted by the parent thread. To do this, the parent thread must have FPU access itself, it then retypes a 512B untyped into an *FPU object* and is granted a new FPU capability by the kernel. The parent thread can then `seL4_TCBBindFPU()` this capability to a single thread. The current model still uses the lazy-FPU, and when faulting, the kernel checks to see if the thread has an FPU capability. If it does, nice, otherwise it will trigger a fault and an exception handler will have to go about cleaning the mess (this is an idea at this point and not implemented).

FPU object

The FPU object *only* stores the main register context, and not the status/control registers (fpsr/fpcr). This is done to keep the FPU object nicely aligned on a power-of-2 boundary. The fpsr and fpcr is instead kept in the TCB.

Cap structure

```
block fpu_cap {
    padding                         64

    field capType                   5
    padding                         11
    field_high capFPUPtr            48
}
```

The pointer to the *FPU register context* is stored in the cap. This is assigned when untyped_retype'ing the FPU object. In future, there may be a field that specifies which TCB the cap is bound to, allowing us to strictly enforce 1:1 FPU-TCB bindings.

fpu_t

`fpu_t` is a struct within the TCB (within arch_tcb_t):

```
typedef struct fpu {
    user_fpu_state_t *fpuState;
    uint32_t fpsr;
    uint32_t fpcr;
} fpu_t;
```

It is important to note that `fpuState` and `capFPUPtr` (in the cap) point to the same object. The reason we need the same pointer in the TCB is because before, the kernel would track which `user_fpu_state_t` was active, and that stored the context, fpsr, and fpcr. However, since we took the fpsr/fpcr outside of `user_fpu_state_t`, we need a way to bundle them again, and so we instead track which `fpu_t` is active.

NOTE: The naming is very confusing right now and will probably change.

TCB

The TCB has been reduced to 1KiB in size from 2KiB. Currently, this is as low as it'd go on aarch64, due to the size of the TCB cnode table, which gets bundled with the main TCB object (under 512B).

Binding to TCB When binding an FPU cap to a thread, the cap is derived from the parent's cspace into the child thread's TCB cnode. The idea is that each FPU cap can only be binded to one TCB, and vice versa. However, this may be the root cause of the memory leak problem.

Memory leak

When running sel4test, each test passes when ran isolated. However, when all the tests are ran at once, we get the following problem:

```
<<seL4(CPU 0) [decodeUntypedInvocation/205 T0xffffff80fffd9200 "sel4test-driver" @419fa0]: Untyped Retype: Insufficient memory (1 * 4096 bytes needed, 0 bytes available).>>
```

The leading theory right now is that after deriving the FPU cap (when binding), there is a copy of the cap lying somewhere, which means the memory cannot be reused as an untyped. The problem is likely to be in the userspace implementation. However, the behaviour of the cap with derivations was not considered originally, and it is possible that there is not enough support in the kernel side implementation to handle the case where the FPU cap gets derived.

Summary of current implementation

- TCB reduced from 2KiB to 1KiB
- FPU object now 512 bytes
- FPU is manually binded to TCB, currently no way to unbind
- FPU faults either lead to graceful execution or a fault some exception handler can handle
- Memory leak 😦