Lab 5: Process management

18307130024 Jimmy Tan

1. `struct proc`

Till now, the `proc` structure contains:

```
struct proc {
    uint64_t sz;             /* Size of process memory (bytes)          */
    uint64_t* pgdir;         /* Page table                              */
    char* kstack;            /* Bottom of kernel stack for this process */
    struct trapframe* tf;    /* Trapframe for current syscall           */
    struct context* context; /* swtch() here to run process             */
    // skip some elements
};
```

This section will discuss whether these pointers should be stored as objects instead.

1.1 `sz`

The `sz` is the user-space memory size of the process.

For the `init` process, the `sz` is `PGSIZE`. For forked processes, the `sz` goes the same with their parent processes. After `exec`, the `sz` is two pages (the first is inaccessible, and the other is used for user stack). Processes can request more user memory.

1.2 Pointers or objects

For page tables, kernel stacks, trapframes, and contexts, being stored as pointers is better, because:

- Storing as objects would heavily increase the size of the bss section.
- Page tables are required to be page aligned.
- Due to locality, making the `proc` well fit in cache would lead to better performance.

- Storing the pointers avoids fixed addresses of the objects, allowing more flexibility.

Apart from performance issues, trapframes should also be stored in the kernel stack instead of `proc` structure, because `alltraps` does not know what the address of the trapframe is. Storing the whole trapframe would increase the complexity of trap handler.

While scheduling, the scheduler is seen as like a process and has its own context. If contexts are stored in `proc` structures, the context of the scheduler shall also be stored in fixed address.

2. Context Switch

2.1 Context

Contexts store callee-saved registers x16 to x30.

Traps are not function calls. In function calls, caller-saved registers would be saved. But traps are asynchronous events, so all common registers should be saved.

The value of registers in trapframes and contexts are not the same. The context is of kernel mode and the trapframe is of the user mode.

2.2 Switch (function `swtch`)

The function pushes the context to the kernel stack, stores the new pointer to the context and loads the context of the other process.

The first argument of `swtch` should be type `**context` because we store the pointer to the context in the `proc` structure and we need to modify the value of the context pointer in the `swtch` function.

3. Process

3.1 `proc_init()`

This function initializes locks related to process management, which are `ptable.lock` and `nextpid.lock`. I add the latter in generating the next process id.

3.2 `proc_alloc()`

This function iterates through the process table to find an unused process.

If not found, return 0.

Otherwise, it generates the kernel stack for this process. The kernel stack involves trapframe, context and so on. Specially, the link register of the context is set as `forkret + 8`.

3.3 `user_init()`

This function initializes the init process for the user space.

It creates a page table, sets the map between the virtual memory and physical memory and loads the code. It also sets the link register in the trapframe to address 0.

3.4 `scheduler()`

This function selects a runnable process and switches to it.

It would never return, iterating the process table and selects a runnable process. Before switching to it, the `ttbr0_el1` register is changed to the page table address of the target process.

3.5 `sched()`

This functions asserts the lock of the process table is held and switches to the scheduler.

3.6 `exit()` and `yield()`

This two functions change the state of the process and switches to the scheduler.

`exit()` changes the state to `ZOMBIE` and `yield()` changes the state to `RUNNABLE`.

In common cases, init process should not exit. In this test, that is OK.

4. System Calls

In `syscall()`, the syscall number is stored in `thiscpu->proc->tf->r0`.

```
switch (proc->tf->r0) {
    case SYS_exec:
        return sys_exec();
    case SYS_exit:
        return sys_exit();
}
```

The `yield()` function is added in the trap function. To test this function, I add three init processes and comment the exit part of the initcode.

```
start:
    ldr     x1, =init
    ldr     x2, =argv
    mov     x0, #SYS_exec
    svc     0x00

exit:
    /* mov     x0, #SYS_exit
    svc     0x00 */
    b       exit
```

```
int
main()
{
    // ...
    user_init();
    user_init();
    user_init();
    // ...
}
```

I also add outputs in scheduler and yield.

```
cprintf("yield: process id %d gives up the cpu %d\n", p->pid, cpuid());
cprintf("scheduler: process id %d takes the cpu %d\n", p->pid, cpuid());
```

Running `make qemu`, the output is like:

```
qemu-system-aarch64 -M raspi3 -nographic -serial null -serial mon:stdio -kernel
obj/kernel8.img
main: [CPU3] is init kernel
main: [CPU2] is init kernel
main: [CPU0] is init kernel
main: [CPU1] is init kernel
Allocator: Init success.
irq_init: - irq init
irq_init: - irq init
irq_init: - irq init
irq_init: - irq init
main: [CPU2] Init success.
main: [CPU0] Init success.
main: [CPU1] Init success.
main: [CPU3] Init success.
scheduler: process id 1 takes the cpu 2
scheduler: process id 2 takes the cpu 0
scheduler: process id 3 takes the cpu 3
sys_exec: executing /init with parameters: sys_exec: executing /init with
parameters: sys_exec: executing /init with parameters: /init /init /init


timer: cpu 2 timer.
timer: cpu 0 timer.
timer: cpu 3 timer.
yield: process id 1 gives up the cpu 2
yield: process id 3 gives up the cpu 3
scheduler: process id 1 takes the cpu 1
scheduler: process id 3 takes the cpu 2
timer: cpu 1 timer.
yield: process id 2 gives up the cpu 0
scheduler: process id 2 takes the cpu 3
yield: process id 1 gives up the cpu 1
scheduler: process id 1 takes the cpu 0
timer: cpu 0 timer.
timer: cpu 2 timer.
timer: cpu 3 timer.
yield: process id 1 gives up the cpu 0
yield: process id 2 gives up the cpu 3
yield: process id 3 gives up the cpu 2
scheduler: process id 1 takes the cpu 1
scheduler: process id 2 takes the cpu 3
```

Printing the parameters of `sys_exec` does not add locks, so the output is out-of-order.

As can be seen in the output, the kernel runs smoothly.

Reversing the code in initcode and main, the output is like:

```
qemu-system-aarch64 -M raspi3 -nographic -serial null -serial mon:stdio -kernel
obj/kernel8.img
```

```
main: [CPU0] is init kernel
main: [CPU3] is init kernel
main: [CPU2] is init kernel
main: [CPU1] is init kernel
Allocator: Init success.
irq_init: - irq init
irq_init: - irq init
irq_init: - irq init
irq_init: - irq init
main: [CPU2] Init success.
main: [CPU3] Init success.
main: [CPU1] Init success.
main: [CPU0] Init success.
scheduler: process id 1 takes the cpu 2
sys_exec: executing /init with parameters: /init
sys_exit: in exit
```