KIV/OS - cvičení č. 6

Martin Úbl

9. srpna 2021

1 Obsah cvičení

- jednoduchý in-memory souborový systém
- FS drivery (UART, GPIO)
- vyhrazení periferií
- propojení se správou procesů
- systémová volání `open`, `read`, `write`, `close`, `ioctl`
- stub RTL pro systémová volání

2 Souborový systém

Náš operační systém bude v budoucnu využívat izolace uživatelských procesů od systémového (kernel) kódu. Stěžejním prvkem tohoto oddělení je právě souborový systém jako prostředek, ve kterém lze organizovat připojená datová média, připojené periferie a jiné prostředky, jejichž správu chceme mít úzce spjatou se správou procesů.

Jelikož tvoříme pouze minimalistický operační systém, který v budoucnu bude systémem reálného času, vytvoříme souborový systém rovněž jednoduchý a přímočarý. Nepotřebujeme VFS v celé svojí kráse – tento systém je pro embedded zařízení a real-time OS příliš složitý a my zdaleka nevyužijeme jeho potenciál. Proto si definujme základní strukturu souborového systému, kdy první částí řetězce cesty identifikujeme „podstrom“ v hierarchii. Tímto identifikátorem může být například:

DEV - připojená periferie (např. UART nebo GPIO)

MNT - připojené datové úložiště (např. SD karta nebo EEPROM)

SYS - systémová nastavení, se kterými může uživatelský proces hýbat (např. povolení nebo zakázání nějaké periferie, pokud to kernel dovolí)

Zbytek bude za dvojtečkou lomítky oddělená cesta, která specifikuje konkrétní zdroj. Jako příklad uveďme:

- `DEV:gpio/10` - označuje GPIO pin číslo 10
- `DEV:uart/0` - označuje UART kanál 0
- `MNT:sd/0/soubor.txt` - označuje soubor `soubor.txt` v kořenovém adresáři na oddílu 0 SD karty
- `SYS:peripherals/uart/0/enable` - označuje soubor, kterým můžeme zakázat nebo povolit UART kanál 0

Toto schéma nám dovoluje pevně definovat položky kořenového adresáře bez nutnosti přehnané dynamické alokace. Zároveň můžeme napsat minimalistický systém „driverů“ pro souborový systém, který umožní připojit jednotlivé části dle dostupných periferií. Pak jen stačí napsat driver pro každou periferii a můžeme periferie ovládat skrze souborový systém.

Začněme strukturou – v kořenovém adresáři zdrojových souborů jádra vytvořme podsložku `fs`. Totéž proveďme u hlavičkových souborů. Tam navíc ještě jako podsložku této úrovně vytvořme složku `drivers`, kam budeme ukládat hlavičkové implementace driverů pro filesystém.

V hlavičkových souborech filesystému vytvořme soubor `filesystem.h`. V něm nyní definujeme konstanty a rozhraní.

Jako první bychom měli stanovit konstanty, které omezí velikosti vybraných struktur a jmen v rámci našeho souborového systému. Znovu je třeba zdůraznit, že píšeme systém pro embedded zařízení, jehož paměť a výpočetní výkon jsou velmi omezené a tak šetříme co možná nejvíce to jde.

Definujme nyní konstanty:

```
constexpr const uint32_t MaxFSDriverNameLength = 16;
constexpr const uint32_t MaxFilenameLength = 16;
constexpr const uint32_t MaxPathLength = 128;
constexpr const uint32_t NoFilesystemDriver = static_cast<uint32_t>(-1);
```

Význam většiny je poměrně zřejmý – omezíme velikost názvu driveru pro souborový systém, pro název souboru, pro délku celé cesty a taktéž definujeme konstantu, která označuje, že daný uzel ve stromu nemá přiřazen žádný driver (později bude zřejmé i proč).

Dále se nám bude hodit i režim otevření souboru:

```
enum class NFile_Open_Mode
{
  Read_Only,
  Write_Only,
  Read_Write,
};
```

Nyní definujme rozhraní (resp. třídu předka) pro soubor. Tato třída, resp. její instance, neopustí hranice jádra! Kód uživatelského procesu ji jen bude moci

označovat pomocí čísla, tzv. file deskriptoru. Veškeré operace bude moci rovněž provádět jen díky tomuto číslu.

```
class IFile
{
  public:
    virtual ~IFile() = default;

    virtual uint32_t Read(char* buffer, uint32_t num) {
      return 0;
    }
    virtual uint32_t Write(const char* buffer, uint32_t num) {
      return 0;
    }
    virtual bool Close() {
      return true;
    }
    virtual bool IOCtl(NIOCtl_Operation dir, void* ctlptr) {
      return false;
    };
};
```

Rozhraní (resp. základní třída) obsahuje poměrně standardní základní sadu metod – čtení, zápis, zavření a modifikace nějakých vlastností. Samozřejmě toho spousty chybí (např. `seek`, ...), což částečně napravíme v dalších cvičeních. Všimněme si rovněž skutečnosti, že zde není metoda `open` – to bude úkolem filesystem driveru, který část režie dle vlastního uvážení přesune do konstruktoru potomka této třídy.

Zbývá už jen rozhraní pro filesystem driver:

```
class IFilesystem_Driver
{
  public:
    virtual void On_Register() = 0;
    virtual IFile* Open_File(const char* path,
                             NFile_Open_Mode mode) = 0;
};
```

Ten pro teď obsahuje pouze dvě metody. První je metoda volaná po registraci, která dovoluje například vytvořit nějaký prvotní stav v paměti. Druhá je pro teď důležitější – ta se bude starat o vytvoření příslušné instance potomka `IFile` dle implementace. Bude tedy ve svém podstromu hledat příslušný zdroj, a pokud ho nalezne a zvládne ho otevřít, vytvoří instanci souboru a vrátí ji vnějšímu kódu.

Námi navržený souborový systém bude mít uzly, které mohou být reprezentovány například následující přepravkou:

```
struct TFS_Tree_Node
```

```
{
  char name[MaxFilenameLength];

  bool isDirectory = false;

  uint32_t driver_idx = NoFilesystemDriver;

  TFS_Tree_Node* parent;
  TFS_Tree_Node* children;
  TFS_Tree_Node* next;

  TFS_Tree_Node* Find_Child(const char* name);
};
```

Každý uzel tedy má nějaké jméno, příznak adresáře a může mít přidělený filesystem driver. Následující položky odpovídají pouze organizaci v paměti – zde jde o obyčejný spojový seznam. Navíc každý prvek ukazuje na rodiče a na prvního potomka.

Rovněž definujme záznam filesystem driveru:

```
struct TFS_Driver
{
  char name[MaxFSDriverNameLength];
  const char* mountPoint;
  IFilesystem_Driver* driver;
};
```

Struktura obsahuje jen název, výchozí „mountpoint“ (zde v uvozovkách, jelikož to není úplně přesný výraz) a odkaz na samotný driver, jehož instance již byla vytvořena (pokud možno staticky, viz dále).

3 Úkol za body

...