KIV/OS - cvičení č. 6

Martin Úbl

24. listopadu 2022

1 Obsah cvičení

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

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).

Ve třídě správce souborového systému dále definujme statické pole filesystem driverů a konstantu indikující jejich počet:

```
static const TFS_Driver gFS_Drivers[];
static const uint32_t gFS_Drivers_Count;
```

Nyní definujme kořenovou položku filesystému a první úroveň (tedy naše pevně dané identifikátory podstromů):

```
TFS_Tree_Node mRoot;
TFS_Tree_Node mRoot_Dev;
TFS_Tree_Node mRoot_Sys;
TFS_Tree_Node mRoot_Mnt;
```

Nyní se přesuňme do implementace (ze které budeme volně přebíhat do hlavičky a dalších modulů). Jako první je třeba souborový systém inicializovat, tedy nastavit první úroveň zanoření v kořenovém adresáři na definované pevné podadresáře. To udělejme v konstruktoru:

```
CFilesystem::CFilesystem()
{
  mRoot.parent = nullptr;
  mRoot.next = nullptr;
  mRoot.children = &mRoot_Dev;
  mRoot.isDirectory = true;
  mRoot.driver_idx = NoFilesystemDriver;
  mRoot.name[0] = '\0';

  mRoot_Dev.parent = &mRoot;
  mRoot_Dev.next = &mRoot_Sys;
  mRoot_Dev.children = nullptr;
  mRoot_Dev.isDirectory = true;
  mRoot_Dev.driver_idx = NoFilesystemDriver;
  strncpy(mRoot_Dev.name, "DEV", 4);

  // ... totez pro MNT a SYS ...
}
```

Implementaci `strncpy` nechám na vás.

Dále je třeba souborový systém inicializovat, a tedy „namountovat“ všechny filesystem drivery a připravit celý strom, kam jen to je možné a vhodné. Filesystem driver necháme „mountovat“ na konkrétní „mountpoint“. Z toho učiníme adresář, a všechny cesty, které do něj vedou, budeme předávat konkrétnímu filesystem driveru, aby je ošetřil dle vlastního uvážení. Dejme tomu tedy, že vyžádáme cestu `DEV:gpio/27`. Filesystem driver GPIO je „mountován“ na bod `DEV:gpio`. Filesystem najde tento bod, zjistí, že je s ním asociován tento driver a předá mu zbytek řetězce (tedy 27). GPIO FS driver pak ví, že jakákoliv číselná

hodnota v rozsahu od 0 do 57 (počet GPIO) je validní a mapuje se na konkrétní GPIO pin. Naparsuje proto číslo 27, vytvoří příslušného potomka `IFile` (např. `CGPIO_File`) a tomu předá v konstruktoru číslo 27.

K samotné inicializaci – následující kód projde všechny FS drivery a „namountuje“ je na jejich danou cestu:

```
void CFilesystem::Initialize()
{
  char tmpName[MaxFilenameLength];
  const char* mpPtr;

  int i, j;

  for (i = 0; i < gFS_Drivers_Count; i++)
  {
    const TFS_Driver* ptr = &gFS_Drivers[i];

    mpPtr = ptr->mountPoint;

    TFS_Tree_Node* node = &mRoot, *tmpNode = nullptr;

    while (mpPtr[0] != '\0')
    {
      for (j = 0; j < MaxPathLength && mpPtr[j] != '\0'; j++)
      {
        if (mpPtr[j] == ':' || mpPtr[j] == '/')
          break;

        tmpName[j] = mpPtr[j];
      }

      tmpName[j] = '\0';
      mpPtr += j + 1;

      tmpNode = node->Find_Child(tmpName);
      if (tmpNode)
        node = tmpNode;
      else
      {
        tmpNode = sKernelMem.Alloc<TFS_Tree_Node>();
        strncpy(tmpNode->name, tmpName, MaxFilenameLength);
        tmpNode->parent = node;
        tmpNode->children = nullptr;
        tmpNode->driver_idx = NoFilesystemDriver;
        tmpNode->isDirectory = true;
        tmpNode->next = node->children;
```

```
        node->children = tmpNode;

        node = tmpNode;
      }
    }

    if (node->driver_idx != NoFilesystemDriver)
      return;

    node->driver_idx = i;

    ptr->driver->On_Register();
  }
}
```

Kód je vcelku přímočarý a snadno pochopitelný – pro všechny FS drivery projde cestu do požadované hloubky a všechny adresáře na cestě vytvoří, pokud již neexistují. Poslednímu článku pak nastaví ID příslušného filesystem driveru, aby bylo jasné, který kontaktovat při požadavku o operaci nad souborovým systémem. Nakonec zavolá metodu `On_Register()`.

V těle je volána metoda `Find_Child`, která je průchodem všech potomků a porovnání jmen na prostou shodu:

```
CFilesystem::TFS_Tree_Node* CFilesystem::TFS_Tree_Node::
    Find_Child(
    const char* name)
{
  TFS_Tree_Node* child = children;

  while (child != nullptr)
  {
    if (strncmp(child->name, name, MaxFilenameLength) == 0)
      return child;

    child = child->next;
  }

  return nullptr;
}
```

Pak potřebujeme metodu `Open`, která bude přejímat cestu k souboru a mód otevření (`NFile_Open_Mode`). Tato metoda obsahuje v podstatě velmi podobný průchod, jako je vidět výše, jen s tím rozdílem, že pokud narazí na neexistující část cesty, vrací chybu. V momentě, kdy narazí na uzel, který má pod správou

nějaký filesystem driver, zavolá jeho metodu `Open` se zbytkem cesty, který ještě parsován nebyl. Implementace této metody je ponechána na čtenáři.

3 FS drivery

Nyní můžeme implementovat konkrétní FS drivery a přidat je do statického pole správce souborového systému. Začneme filesystem driverem pro UART. V hlavičkových souborech pro FS drivery vytvořme soubor `uart_fs.h`. Tady vytvořme minimalistický UART FS driver:

```
class CUART_FS_Driver : public IFilesystem_Driver
{
  public:
    virtual void On_Register() override
    {
    }

    virtual IFile* Open_File(const char* path,
                             NFile_Open_Mode mode) override
    {
      int channel = atoi(path);
      if (channel != 0)
        return nullptr;

      CUART_File* f = new CUART_File(channel);

      return f;
  }
};
```

Tady ještě něco chybí. K tomu se pak vraťme v další kapitole. Jak je vidět, je kód velmi jednoduchý – tento driver nevyžaduje žádnou inicializaci, a metoda pro otevření jen parsuje číslo ze vstupu a porovnává ho s nulou – to je totiž jediný kanál UARTu, který máme k dispozici. Pokud se tohle povede, je instancována třída `CUART_File` (kterou představíme záhy) a je vrácena. Jak vidíte v kódu, použili jsme klíčové slovo `new`, ale to my víme, že souvisí s dynamickou alokací nějakými standardními metodami. Ty my v jádře nemáme a musíme je dodefinovat. Vrátíme se k nim na konci tohoto cvičení (poslední kapitola).

Nyní ke třídě `CUART_File`. Ta bude pro teď podporovat pouze zápis. Níže pak implementujeme ještě `ioctl` a v některém z dalších cvičení možná i čtení (až budeme umět blokovat proces nad čtením ze souboru). Soubor by měl mít indikaci toho, zda byl uzavřený, aby nedošlo k opětovému uzavření periferie (z důvodu např. vyhrazení, viz dále). Implementace pak může vypadat například takto:

```
class CUART_File : public IFile
{
  private:
    int mChannel;

  public:
    CUART_File(int channel)
      : mChannel(channel)
    {
    }

    ~CUART_File()
    {
      Close();
    }

    virtual uint32_t Write(const char* buffer,
                           uint32_t num) override
    {
      if (num > 0 && buffer != nullptr)
      {
        if (mChannel == 0)
        {
          sUART0.Write(buffer, num);
          return num;
        }
      }

      return 0;
    }

    virtual bool Close() override
    {
      if (mChannel < 0)
        return false;

      mChannel = -1;

      return true;
    }
};
```

Tento hlavičkový soubor budeme includovat pouze a jen v jednom místě – v souboru `filesystem_drivers.cpp`. Nemusíme se tedy bát na konec tohoto souboru

vložit instancování FS driveru:

```
CUART_FS_Driver fsUART_FS_Driver;
```

Teď definujme obsah souboru `filesystem_drivers.cpp` – bude obsahovat pouze pole FS driverů a statickou inicializaci konstanty počtu driverů:

```
#include <fs/filesystem.h>
#include <fs/drivers/uart_fs.h>

const CFilesystem::TFS_Driver CFilesystem::gFS_Drivers[] =
{
  { "UART_FS", "DEV:uart", &fsUART_FS_Driver },
};

const uint32_t CFilesystem::gFS_Drivers_Count =
  sizeof(CFilesystem::gFS_Drivers)
   / sizeof(CFilesystem::TFS_Driver);
```

4 Vyhrazení periferií

Jistě každý z nás narazil v nějakém „velkém“ operačním systému na situaci, kdy jsme chtěli otevřít nějaký soubor (použít nějakou periferii), ale bylo nám odpovězeno chybovým kódem a hláškou „Device is busy“ (ve Windows něco jako „Device is currently in use“.) – to proto, že zařízení bylo právě otevřené a používané jiným procesem, který měl na něj exkluzivní práva. To je často z podstaty věci – typicky nechceme, aby jedno zařízení používalo více procesů naráz, a když ano, máme na to patřičné mechanismy, které procesy synchronizují (např. spooling, fronty, kanály, ...).

V případě našeho systému bude stačit, když jednotlivým driverům periferií implementujeme metody, které dovedou příslušné zařízení (a jeho kanál, pin, ...) uzamknout a odemknout. Pro UART stačí, když implementujeme metodu `Open` a `Close`, kdy metoda `Open` bude vnitřně ověřovat, zda je zařízení již otevřené, a pokud ne, příznak nastaví. Vracet bude vždy příznak toho, zda se zařízení povedlo či nepovedlo zabrat (zda již bylo otevřené nebo ne). Metoda `Close` pak bude tento příznak odnastavovat.

Pak upravme kód UART FS driveru – do metody `Open_File` před samotné vytvoření souboru vložme tyto řádky:

```
if (!sUART0.Open())
  return nullptr;
```

Do třídy `CUART_File` pak ještě doplňme do metody `Close` volání `sUART0.Close()` a vše by mělo být připraveno.

Stejně tak musíme myslet na podobné vyhrazení periferií i v ostatních případech.

5 Propojení se správou procesů

Každý proces si bude moci otevřít soubory dle potřeby. Do PCB musíme proto vložit další položku, která bude představovat otevřené soubory. Pro naše potřeby bude stačit, když půjde o pole ukazatelů na `IFile`, které bude mít statickou velikost několika málo položek – např. 16.

Struktura PCB a daná konstanta nyní budou vypadat takto:

```
constexpr uint32_t Max_Process_Opened_Files = 16;

struct TTask_Struct
{
  TCPU_Context cpu_context;
  unsigned int pid;
  NTask_State state;
  unsigned int sched_counter;
  unsigned int sched_static_priority;
  IFile* opened_files[Max_Process_Opened_Files];
};
```

Index v poli `opened_files` bude odpovídat souborovému deskriptoru, který bude poskytnut procesu při otevření souboru (tedy jako výsledek budoucího systémového volání `open()`).

Do správce procesů přidejme způsob, jakým mapovat otevřený soubor (typu `IFile*`) do PCB. Vytvořme proto metody `Map_File_To_Current(IFile* file)` a `Unmap_File_From_Current(uint32_t handle)`. Mapovací funkce bude vracet vždy indikátor úspěchu. Vnitřně bude hledat první nepřiřazený slot v poli `opened_files` v současnosti naplánovaného procesu, tam soubor umístí a vrátí index. Pokud již proces překročil maximální počet otevřených souborů, vrací chybový kód (např. -1). Metoda pro odmapování bude na konkrétním indexu odmapovávat soubor a uvolní daný slot. Nebude se však pokoušet soubor zavřít – v tento moment již instance souboru nemusí ani existovat.

Tyto metody budeme volat z obsluhy systémových volání.

6 Systémová volání

Systémové volání je hlavním prostředkem pro komunikaci uživatelského procesu s jádrem. Prostřednictvím něj budeme žádat o zdroje, soubory, ovládat periferie, a tak podobně.

Na systémech založených na ARM architektuře je systémové volání vyvoláno instrukcí `SVC` („supervisor call“; někdy označováno starším názvem `SWI`). Tato instrukce může být volána s jedním operandem. Ten je kódován do dolních třech bajtů instrukce a procesor jej nikam nekopíruje (např. do nějakého registru). Musíme si jej proto z kódované instrukce extrahovat sami.

V předchozích cvičeních jsme implementovali obsluhu přerušení a pro obsluhu systémových volání jsme ponechali zatím jen prázdný stub – nepoužívali jsme jej. Nyní jej už potřebovat budeme, a tak implementujme první úroveň obsluhy v assembly, abychom korektně zvládli dekódovat parametr instrukce, uložit a obnovit kontext procesu a zavolat druhou úroveň obsluhy v C/C++ kódu.

Systémové volání bude předávat parametry a návratové hodnoty výhradně prostřednictvím registrů. Částečně využijme volací konvenci ARM, a pro systémové volání vyhraďme registry `r0`, `r1` a `r2`. Registr `r3` nechme volný, abychom měli kam dekódovat spodní část instrukce a tedy samotnou službu, kterou se snažíme vyvolat. Systémové volání bude výsledky vracet v registrech `r0` a `r1`.

První úroveň tedy může vypadat třeba takto:

```
.global _internal_software_interrupt_handler
software_interrupt_handler:
  stmfd sp!,{r2-r12,lr}
  ldr r3,[lr,#-4]
  bic r3,r3,#0xff000000
  bl _internal_software_interrupt_handler
  mov r2, r0
  ldr r0, [r2, #0]
  ldr r1, [r2, #4]
  ldmfd sp!, {r2-r12,pc}^
```

Jak je vidět, nejprve uložíme registry procesu a návratovou adresu, a pak načteme do registru `r3` kus paměti, který je o 4 bajty před současným obsahem registru `lr`. Jde tedy o 4 bajty, které představují zakódovanou instrukci, která vyvolala přerušení – `lr` ukazuje na následující instrukci (tedy kam se vrátit). Z této sekvence bajtů vymaskujeme poslední 3 bajty a ty nám označují službu, kterou se snažíme vyvolat. Následně zavoláme obsluhu v C++ kódu, do které se jako parametry „automaticky“ předají registry `r0`, `r1`, `r2` a `r3`.

Ještě než se pustíme do implementace C++ části, je nutné si trochu rozmyslet, jak vlastně budeme mít rozdělená systémová volání na „obslužné jednotky“ (tzv. *facility*) a jednotlivé služby v nich. Identifikátor, který předala instrukce ve svém těle by sice mohl být 24-bitový, ale instrukční sada definuje, že lze do těla univerzálně kódovat pouze 8-bitové číslo. To je koneckonců poměrně logické – instrukce musí být kompatibilní se všemi režimy procesoru, mj. i s Thumb režimem, který instrukce kóduje pouze na 2 bajty. Thumb režim je dosti ceněným nástrojem v ARM světě, a tak mu nelze odepřít systémová volání – počítejme tedy s tím, že toto číslo má pouze 8 bitů.

Rozdělme tedy identifikátor na dvě části – vrchní dva bity nám budou stačit pro obslužnou jednotku (facility), jelikož těch moc nebude. V zásadě nám bude stačit identifikátor pro „procesy“ a „filesystem“. Dolních 6 bitů tedy musí stačit pro číslo služby, které může být dále rozmělněno pomocí obsahu registrů. Máme tedy prostor pro 4 facility a 64 služeb v každé z nich.

Nejprve si tedy definujme hlavičkový soubor v adresáři pro procesové hlavičky `swi.h`, ve kterém definujeme identifikátory, struktury a služby, které budeme záhy implementovat:

Začněme kontejnerem pro výsledek systémového volání – z důvodu vícečetného zanoření předávejme obsah odpovědi v přepravce, z níž nastavíme příslušné registry až těsně před přechodem zpět do procesu. Zamezíme tak nechtěnému přemazání.

```
struct TSWI_Result
{
  uint32_t r0;
  uint32_t r1;
};
```

Dále je dobré specifikovat návratové kódy pro úspěch a neúspěch:

```
enum class NSWI_Result_Code
{
  OK   = 0,
  Fail = 1,
};
```

Pak samotný výčet facilities:

```
enum class NSWI_Facility
{
  Process    = 0b00,
  Filesystem = 0b01,
};
```

A pak už definujeme jen služby včetně nějakých doprovodných komentářů, co která služba provádí, co v jakém registru očekává a tak podobně. Začněme process facility, tam toho máme zatím nejméně:

```
enum class NSWI_Process_Service
{
  // Vraci PID procesu
  // IN:  -
  // OUT: r0 = PID procesu
```

```
  Get_PID          = 0,
};
```

A pak přijde na řadu filesystem facility:

```
enum class NSWI_Filesystem_Service
{
  // Otevre soubor
  // IN:  r0 = ukazatel na retezec identifikujici soubor,
  //      r1 = rezim otevreni souboru
  // OUT: r0 = handle otevreneho souboru nebo Invalid_Handle
  Open             = 0,

  // Precte ze souboru
  // IN:  r0 = handle otevreneho souboru,
  //      r1 = buffer,
  //      r2 = pocet znaku/velikost bufferu
  // OUT: r0 = pocet prectenych znaku
  Read             = 1,

  // Zapise do souboru
  // IN:  r0 = handle otevreneho souboru,
  //      r1 = buffer,
  //      r2 = pocet znaku/velikost bufferu
  // OUT: r0 = pocet zapsanych znaku
  Write            = 2,

  // Zavre soubor
  // IN:  r0 = handle otevreneho souboru
  // OUT: r0 = indikator uspechu (NSWI_Result_Code)
  Close            = 3,

  // Ziska/zmeni parametry souboru dle jeho typu
  // IN:  r0 = handle otevreneho souboru,
  //      r1 = identifikator operace (NIOCtl_Operation),
  //      r2 = ukazatel na strukturu s nastavenim (specificka pro
     soubor)
  // OUT: r0 = indikator uspechu (NSWI_Result_Code)
  IOCtl            = 4,
};
```

Do správce procesů nyní implementujme základ metod pro ošetření jednotlivých služeb v rámci každé facility. Prototypy metod tedy budou vypadat nějak takto:

```
void Handle_Process_SWI(NSWI_Process_Service svc_idx,
     uint32_t r0, uint32_t r1, uint32_t r2, TSWI_Result& target);
void Handle_Filesystem_SWI(NSWI_Filesystem_Service svc_idx,
     uint32_t r0, uint32_t r1, uint32_t r2, TSWI_Result& target);
```

V `interrupt_handler.cpp` kódu pak vytvořme druhou úroveň obsluhy, která tyto metody bude volat:

```
static TSWI_Result _SWI_Result;

extern "C" TSWI_Result* _internal_software_interrupt_handler(
  uint32_t register r0, uint32_t register r1,
  uint32_t register r2, uint32_t register service_identifier)
{
  NSWI_Facility facility =
    static_cast<NSWI_Facility>(service_identifier >> 6);

  const uint8_t service_id_base = service_identifier & 0x3F;

  switch (facility)
  {
    case NSWI_Facility::Process:
      sProcessMgr.Handle_Process_SWI(
          static_cast<NSWI_Process_Service>(service_id_base),
          r0, r1, r2, _SWI_Result);
      break;
    case NSWI_Facility::Filesystem:
      sProcessMgr.Handle_Filesystem_SWI(
          static_cast<NSWI_Filesystem_Service>(service_id_base),
          r0, r1, r2, _SWI_Result);
      break;
  }

  return &_SWI_Result;
}
```

Nyní už zbývá jen implementovat příslušné facility a služby.

Procesová facility bude velmi jednoduchá; volitelně pak můžeme přidat indikaci chyby, ale pro teď ponechme jen holé handlery:

```
switch (svc_idx)
{
  case NSWI_Process_Service::Get_PID:
    target.r0 = mCurrent_Task_Node->task->pid;
```

```
    break;
}
```

Filesystem facility je rovněž poměrně přímočará:

```
switch (svc_idx)
{
  case NSWI_Filesystem_Service::Open:
  {
    target.r0 = Invalid_Handle;

    IFile* f = sFilesystem.Open(reinterpret_cast<const char*>(r0)
    , static_cast<NFile_Open_Mode>(r1));
    if (!f)
      return;

    target.r0 = Map_File_To_Current(f);

    if (target.r0 == Invalid_Handle)
    {
      f->Close();
      delete f;
    }
    break;
  }
  case NSWI_Filesystem_Service::Read:
  {
    target.r0 = 0;

    if (r0 > Max_Process_Opened_Files || !mCurrent_Task_Node->
    task->opened_files[r0])
      return;

    target.r0 = mCurrent_Task_Node->task->opened_files[r0]->Read(
    reinterpret_cast<char*>(r1), r2);
    break;
  }
  case NSWI_Filesystem_Service::Write:
  {
    target.r0 = 0;

    if (r0 > Max_Process_Opened_Files || !mCurrent_Task_Node->
    task->opened_files[r0])
      return;
```

```
    target.r0 = mCurrent_Task_Node->task->opened_files[r0]->Write
    (reinterpret_cast<const char*>(r1), r2);
    break;
  }
  case NSWI_Filesystem_Service::Close:
  {
    if (r0 > Max_Process_Opened_Files || !mCurrent_Task_Node->
    task->opened_files[r0])
      return;

    target.r0 = mCurrent_Task_Node->task->opened_files[r0]->Close
    ();
    Unmap_File_Current(r0);

    break;
  }
  case NSWI_Filesystem_Service::IOCtl:
  {
    if (r0 > Max_Process_Opened_Files || !mCurrent_Task_Node->
    task->opened_files[r0])
      return;

    target.r0 = mCurrent_Task_Node->task->opened_files[r0]->IOCtl
    (static_cast<NIOCtl_Operation>(r1), reinterpret_cast<void*>(
    r2));
    break;
  }
}
```

Tím by měla být hlavní část systémových volání implementována.

7 Systémové volání IOCtl

Některá zařízení (a potažmo jejich soubory) mohou vyžadovat dodatečná nastavení, která nelze specifikovat při jejich otevírání nebo zápisem do nich. Mezi dva dominantní způsoby, jak se s tím vypořádat, patří buď systémové volání `ioctl` (a jím podobné) nebo definice protokolu, který slouží jako mezivrstva v souborovém systému a samotný „datový“ přenos je obsažen jako jedna ze zpráv tohoto protokolu.

Druhému způsobu se budeme více věnovat v dalším cvičení. Pro teď se zaměřme na systémové volání `ioctl` a jeho aplikaci na UART – ten má totiž nějaké parametry, které je nutné nastavit, aby přenos fungoval (baudovou rychlost, velikost znaku, ...).

Obvyklou implementací `ioctl` je takové volání, které přejímá souborový deskriptor, kód požadavku (specifický pro zařízení) a ukazatel na strukturu, která je s daným kódem požadavku asociovaná. Pro teď si definujme dva víceméně generické kódy:

```
enum class NIOCtl_Operation
{
  Get_Params      = 0,
  Set_Params      = 1,
};
```

Tyto kódy označují takové operace, které mají vyzískat aktuální nastavení nebo naopak přepsat nastavení dodanou sadou parametrů.

U generického předka souboru `IFile` již máme vytvořenou metodu `IOCtl`. Stačí pouze její tělo pro konkrétní soubory implementovat. Začněme tedy u definice struktur pro UART a jeho `ioctl` operace. Jako přepravku s nastavením vytvořme v hlavičkovém souboru jeho driveru strukturu:

```
struct TUART_IOCtl_Params
{
  NUART_Char_Length char_length;
  NUART_Baud_Rate baud_rate;
};
```

Pak jen stačí v implementaci `CUART_File` implementovat zmíněnou metodu:

```
virtual bool IOCtl(NIOCtl_Operation dir, void* ctlptr) override
{
  if (dir == NIOCtl_Operation::Get_Params)
  {
    TUART_IOCtl_Params* params = reinterpret_cast<
    TUART_IOCtl_Params*>(ctlptr);
    params->baud_rate = sUART0.Get_Baud_Rate();
    params->char_length = sUART0.Get_Char_Length();
    return true;
  }
  else if (dir == NIOCtl_Operation::Set_Params)
  {
    TUART_IOCtl_Params* params = reinterpret_cast<
    TUART_IOCtl_Params*>(ctlptr);
    sUART0.Set_Baud_Rate(params->baud_rate);
    sUART0.Set_Char_Length(params->char_length);
    return true;
  }
  return false;
```

```
}
```

A pak pochopitelně odpovídajícím způsobem implementovat v driveru metody `Get_Baud_Rate()`, `Get_Char_Length()` a jejich `Set_*` varianty.

8 RTL

Systémová volání definovaná máme, ale abychom je mohli bez problémů a přehledně volat, je téměř nutností implementovat minimalistický obal nad vyvoláním příslušné facility a služby. Založme si proto podprojekt běhové knihovny (RTL), pokud už jsme to nestihli během minulých cvičení. Ten se bude linkovat staticky ke všemu, co bude určeno k běhu na našem systému.

Vytvořme pak hlavičkový soubor `stdfile.h` a odpovídající implementaci `stdfile.cpp`. V nich vytvořme pro každé systémové volání jednu funkci včetně parametrů, které má přejímat.

Uvnitř volání je nutné nastavit vstupní registry, vyvolat instrukci `svc` s příslušným parametrem a postarat se o předání návratové hodnoty. Pro názornost proveďme implementaci v inline assembly, tedy vnořeném assembly do C/C++ kódu.

Pro volání `open` tedy může funkce vypadat třeba takto:

```
uint32_t open(const char* filename, NFile_Open_Mode mode)
{
  uint32_t file;

  asm volatile("mov␣r0,␣%0" : : "r" (filename));
  asm volatile("mov␣r1,␣%0" : : "r" (mode));
  asm volatile("svc␣64");
  asm volatile("mov␣%0,␣r0" : "=r" (file));

  return file;
}
```

Inline assembly lze v tomto případě propojit vcelku elegantně – pokud chceme obsah nějaké proměnné přesunout do registru, stačí předat její identifikátor v příslušné sekci inline assembly direktivy, a kompilátor, jelikož stejně bude kompilovat kód do strojového kódu, jehož je assembly přímým přepisem, jen nahradí parametr skutečným umístěním identifikátoru.

Tímto způsobem do `r0` a `r1` přesuneme příslušné parametry (s největší pravděpodobností tam už ale byly, protože ARM volací konvence je tam přesně takto uložila), zavoláme systémové volání `svc 64` (64 = facility 1, služba 0, tedy `open`).

Obdobně můžeme implementovat i zbytek:

```
uint32_t read(uint32_t file, char* const buffer, uint32_t size)
{
  uint32_t rdnum;

  asm volatile("mov␣r0,␣%0" : : "r" (file));
  asm volatile("mov␣r1,␣%0" : : "r" (buffer));
  asm volatile("mov␣r2,␣%0" : : "r" (size));
  asm volatile("swi␣65");
  asm volatile("mov␣%0,␣r0" : "=r" (rdnum));

  return rdnum;
}

uint32_t write(uint32_t file, const char* buffer, uint32_t size)
{
  uint32_t wrnum;

  asm volatile("mov␣r0,␣%0" : : "r" (file));
  asm volatile("mov␣r1,␣%0" : : "r" (buffer));
  asm volatile("mov␣r2,␣%0" : : "r" (size));
  asm volatile("swi␣66");
  asm volatile("mov␣%0,␣r0" : "=r" (wrnum));

  return wrnum;
}

void close(uint32_t file)
{
  asm volatile("mov␣r0,␣%0" : : "r" (file));
  asm volatile("swi␣67");
}

uint32_t ioctl(uint32_t file, NIOCtl_Operation operation, void*
    param)
{
  uint32_t retcode;

  asm volatile("mov␣r0,␣%0" : : "r" (file));
  asm volatile("mov␣r1,␣%0" : : "r" (operation));
  asm volatile("mov␣r2,␣%0" : : "r" (param));
  asm volatile("swi␣68");
  asm volatile("mov␣%0,␣r0" : "=r" (retcode));

  return retcode;
}
```

```
uint32_t getpid()
{
  uint32_t pid;

  asm volatile("swi␣0");
  asm volatile("mov␣%0,␣r0" : "=r" (pid));

  return pid;
}
```

9 Operátory new a delete

Jak jsme zmínili v předchozích kapitolách, budeme chtít alokovat paměť pomocí operátorů `new` a dealokovat pomocí `delete`. V C++ mají oba tyto operátory několik variant – „obyčejnou“ pro skalární alokaci, pro alokací polí a tzv. placement `new` variantu. V zásadě ale pro teď nepotřebujeme nic složitého – jen aby používaly naši implementaci kernelové haldy a placement varianty byly čistě transparentní. Doplňme proto soubor `kernel_heap.h` o následující inline implementace:

```
inline void* operator new(uint32_t size)
{
  return sKernelMem.Alloc(size);
}

inline void *operator new(uint32_t, void *p)
{
  return p;
}

inline void *operator new[](uint32_t, void *p)
{
  return p;
}

inline void operator delete(void* p)
{
  sKernelMem.Free(p);
}

inline void operator delete(void* p, uint32_t)
{
```

```
  sKernelMem.Free(p);
}

inline void  operator delete  (void *, void *)
{
}

inline void  operator delete[](void *, void *)
{
}
```

V kontextu vývoje OS má placement varianta význam především v momentě, kdy potřebujeme na předem jasném místě vykonstruovat objekt daného typu. Například již zmíněné memory-mapped I/O mají pevně danou adresu, díky které je možné ovládat periferie a jiné. V tomto případě můžeme „alokovat“ tuto paměť pomocí placement operátoru a jde vlastně o ekvivalentní operaci k `reinterpret_cast` na daný typ (jen možná trochu přehlednější):

```
uint32_t* peripheral_regs =
    new (hal::Peripheral_Base) uint32_t[32];

uint32_t* peripheral_regs =
    reinterpret_cast<uint32_t*>(hal::Peripheral_Base);
```

Teď ale máme situaci, kdy potřebujeme například na předem alokované paměti pro nějaký objekt skutečně provést konstrukci objektu (zavolat konstruktor, inicializovat atributy, ...). Pak nám nezbývá než použít placement `new` operátor a nad takto alokovanou pamětí ho zavolat:

```
void* mem = sKernelHeap.Alloc(sizeof(CObject));

CObject* obj = new (mem) obj(1, 2, "hello");
```

Bude ale bohatě stačit, když přepíšeme klasický operátor `new` (a odpovídající `delete`), jelikož přesměrováváme jeho implementaci na naši implementaci kernelové haldy.

10 Testovací program

Teď můžeme vše otestovat. Upravme proto jeden z procesů, aby prováděl tento kód:

```
volatile int i;

const char* msg = "Hello!\r\n";

uint32_t f = open("DEV:uart/0", NFile_Open_Mode::Read_Write);

TUART_IOCtl_Params params;
params.baud_rate = NUART_Baud_Rate::BR_115200;
params.char_length = NUART_Char_Length::Char_8;
ioctl(f, NIOCtl_Operation::Set_Params, &params);

while (true)
{
  write(f, msg, strlen(msg));

  for (i = 0; i < 0x80000; i++)
    ;
}

close(f);
```

Výsledkem by mělo být cyklické vypisování řetězce `Hello!` na UART konzoli.

11 Úkol za body

Implementujte podporu pro miniUART filesystem driver – jednak umožněte zápis, ale zároveň i čtení pomocí mezilehlého bufferu. Pokud jste neimplementovali čtení z UARTu v minulých úlohách, předpokládejte, že do tohoto bufferu by něco zapisovalo (přerušení).

Buffer realizujte jako kruhový a implementujte ho v driveru UARTu. Filesystem by pak měl z tohoto bufferu umět číst dostupné znaky. Pro teď předpokládejte neblokující čtení, čili pokud v bufferu nic není, indikujte to patřičným způsobem (návratová hodnota) do uživatelského procesu.