KIV/OS - cvičení č. 9

Martin Úbl

3. prosince 2021

1 Obsah cvičení

- awaitable soubory (Wait a Notify)
- GPIO přerušení
- spinlock, mutex, semafor
- podmínková proměnná
- pojmenovaná roura
- EDF plánovač (soft real-time)

2 Úlohy, deadline, awaitable soubory

V typickém systému reálného času obvykle chceme reagovat na vnější podněty, často z důvodu jeho nasazení na embedded zařízení, které na tyto podněty reaguje příslušnou akcí (která musí být vykonána do určitého času – deadline). Tyto akce provádí jednotlivé procesy, které jsou tomuto účelu vyhrazené. Ty čekají na příchod podnětu, provedou příslušnou akci, opět se uspí a to stále dokola opakují. Ve vyšší abstrakci pak lze hovořit o tom, že příchodem podnětu vzniká *task*, který musí být zpracován typicky právě do nějakého shora omezeného času.

2.1 Model úloh

Každý systém reálného času může tyto úlohy implementovat různými způsoby. Třeba může na tyto podněty souhrnně čekat jeden „rodičovský“ proces, a ten po příchodu vytvoří proces specifický, který danou úlohu zpracuje. Vzhledem k tomu, že primárním požadavkem jakéhokoliv systému reálného času je minimální odezva, může toto řešení být nevyhovující. Prakticky proto můžeme procesy pro každou úlohu vytvořit rovnou a každý individuálně nechat čekat na daný podnět. V našem systému zvolíme tento druhý model.

V momentě, kdy podnět přijde, je třeba nastavit deadline. Deadline, tedy horní omezení času zpracování daného podnětu, je třeba nastavit v momentě, kdy task vzniká. Prakticky v námi zvoleném modelu jde o moment probuzení procesu.

Deadline je důležitým parametrem plánování procesu – o plánování procesů na základě deadline více pojednává jedna z posledních kapitol.

2.2 Rozhraní

Zmínili jsme, že proces bude čekat na podnět – zbývá tedy jen vyřešit konkrétní způsob, jakým se bude proces uspávat, a jakým bude probuzen. V kontextu našeho systému se nabízí pro tento mechanismus čekání využít souborový systém, k němuž jsme již v minulosti vytvořili rozhraní v podobě systémových volání `open`, `close`, `read` a `write`. To by ve své podstatě stačilo i pro tyto účely za předpokladu, že dovolíme `read` blokovat. Pro lepší definici sémantiky však dodefinujme dvě další operace – `wait` a `notify`.

Operace `wait` uspí proces nad souborem, pokud to bude nutné. Sémanticky tedy bude proces čekat na zdroj, až bude dostupný. Pokud zdroj dostupný je, proces blokovat (uspávat) nebude, zdroj zabere a okamžitě se vrací. Operace bude mít návratovou hodnotu typu *bool*, kdy *true* bude znamenat, že se čekání povedlo (a zdroj je zabraný) a *false* bude značit neúspěch.

Operace `notify` notifikuje čekající procesy nad souborem, pokud nějaké jsou. Sémantika bude závislá na druhu souboru – buď může jít čistě o notifikaci procesu (např. podmínková proměnná), nebo může jít o inkrementaci čítače zdrojů (např. roura, semafor). Operace bude mít návratovou hodnotu znamenající skutečný počet notifikovaných procesů (resp. zdrojů).

Toto rozhraní je nyní potřeba integrovat do již existujících entit a obsluh. Konkrétně půjde o rozhraní `IFile` a obsluhu systémových volání.

2.3 Awaitable soubor

Rozhraní `IFile` je pro budoucí rozšíření potřeba připravit – nyní bude možné nad souborem čekat a být notifikován, a to nutně znamená, že vyžadujeme frontu čekajících procesů. Tímto se nám z rozhraní stane již nějaká forma abstraktní třídy.

Rozšiřme proto `IFile` o privátní a protected členy:

```
private:
    struct TWaiting_Task
    {
        uint32_t pid;
        TWaiting_Task* next;
        TWaiting_Task* prev;
    };

    TWaiting_Task* mWaiting_Tasks = nullptr;
```

```
protected:
    void Wait_Enqueue_Current();
```

Zde je poměrně zřejmé, co jsme tím mysleli – definovali jsme si prvek fronty a metodu, která dovolí proces do fronty vložit. Samotné uspávání je pak věc odlišná a bude ji řešit volající. Je ale pravda, že pro všechny naše use-cases budeme vždy s vložením do fronty potřebovat proces i uspat.

Implementujme pak metodu `Wait_Enqueue_Current()`:

```
void IFile::Wait_Enqueue_Current()
{
    // TODO: tady budeme zamykat zamek, az nejaky budeme mit

    TWaiting_Task* task = new TWaiting_Task;
    task->pid = sProcessMgr.Get_Current_Process()->pid;
    task->prev = nullptr;
    task->next = nullptr;

    if (!mWaiting_Tasks)
        mWaiting_Tasks = task;
    else
    {
        mWaiting_Tasks->prev = task;
        task->next = mWaiting_Tasks;
        mWaiting_Tasks = task;
    }

    // TODO: tady zase budeme zamek odemykat
}
```

Nyní může kterákoliv implementace `IFile` vložit v současnosti naplánovaný proces do fronty čekajících procesů. Kód obsahuje pár „TODO“ bloků – to proto, že se jedná o kritickou sekci, a v budoucnu budeme potřebovat na těchto místech získávat a vracet zámek.

2.4 Rozšíření IFile

Třídu `IFile` je potřeba rozšířit o rozhraní pro volání `Wait` a `Notify`. Volání `Wait` bude vždy specifické pro každý druh souboru – nelze genericky určit, zda bude potřeba proces uspat nebo ne. Metoda `Notify` však může mít společný základ – ta bude vždy probouzet N procesů z fronty, pokud nepůjde o specifický druh souboru. Takový si ale může metodu přepsat a chování specifikovat jinak.

Nejprve do `IFile` přidejme stub metody `Wait` a `Notify`:

```
virtual bool Wait(uint32_t count) { return true; };
virtual uint32_t Notify(uint32_t count);
```

A metodu `Notify` můžeme implementovat:

```
uint32_t IFile::Notify(uint32_t count)
{
    // TODO: zamknout zamek

   TWaiting_Task* tmp;
   TWaiting_Task* itr = mWaiting_Tasks;
   while (itr && itr->next)
   itr = itr->next;

   uint32_t notified_count = 0;
   while (itr && notified_count < count)
   {
      sProcessMgr.Notify_Process(itr->pid);

      tmp = itr;

      itr = itr->prev;
      delete tmp;
      notified_count++;

      if (itr)
      itr->next = nullptr;
      else
      {
         mWaiting_Tasks = nullptr;
         break;
      }
   }

    // TODO: odemknout zamek

   return notified_count;
}
```

Implementace je opět poměrně přímočará – notifikujeme soubory po jednom, prostřednictvím správce procesů je probouzíme a odebíráme je z fronty čekajících procesů.

2.5 Rozšíření obsluhy systémových volání

Nyní rozšíříme systémová volání a jejich obsluhu. Definujme konstanty do výčtového typu `NSWI_Filesystem_Service`:

```
// Notifikace souboru
// IN:  r0 = handle otevreneho souboru, r1 = pocet zdroju
// OUT: r0 = pocet skutecne notifikovanych zdroju
Notify          = 5,

// Cekani na udalost nad souborem (nejaky zapis, notifikace, ...)
// IN:  r0 = handle otevreneho souboru, r1 = pocet zdroju
// OUT: r0 = indikator uspechu (NSWI_Result_Code)
Wait            = 6,
```

Implementace pak bude jen transparentně volat příslušné metody v otevřeném souboru:

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

   target.r0 = mCurrent_Task_Node->task->opened_files[r0]
                       ->Notify(r1);
   break;
}
case NSWI_Filesystem_Service::Wait:
{
   if (r0 > Max_Process_Opened_Files
       || !mCurrent_Task_Node->task->opened_files[r0])
      return;

   target.r0 = mCurrent_Task_Node->task->opened_files[r0]
                       ->Wait(r1);
   break;
}
```

Nyní tedy v podstatě stačí, když otevřeme příslušný soubor, který implementuje `Wait` dle svých interních pravidel (mutex, roura, ...) a zbytek začne fungovat téměř automaticky, nebo s minimálním úsilím navíc.

Základ našeho systému tedy lze ovládat pomocí relativně malé sady systémových volání: `open`, `close`, `read`, `write`, `wait` a `notify`.

Doplňme pro teď ještě jeden relevantní kus implementace. Každý `wait` syscall, respektive volání `Wait` metody třídy `IFile` může proces blokovat. V tomto případě se jedná terminologicky o převedení procesu do stavu „blokovaný“ a přeplánování. Definujme proto navíc ještě metodu `Block_Current_Process()` ve správci procesů:

```
void CProcess_Manager::Block_Current_Process()
{
   TTask_Struct* cur = Get_Current_Process();
   cur->state = NTask_State::Blocked;
   Schedule();
}
```

A pochopitelně definujme i výčtovou hodnotu `Blocked` ve výčtovém typu `NTask_State`.

2.6 Sleep syscall

Z blíže zatím nespecifikovaných důvodů budeme potřebovat i systémové volání `sleep`. Tyto důvody budou uvedeny v kapitole s plánovačem, jelikož definice souvisí (bude souviset) s určitým druhem tasků.

Pro úspěšnou implementaci systémového volání `sleep` potřebujeme donutit časovač uchovávat počet tiků, definovat nový stav procesu, rozšířit PCB o čas, ve kterém má být proces probuzen, a pak pochopitelně definovat a implementovat obsluhu systémového volání a upravit plánovač, aby uměl procesy probouzet.

Nejprve rozšiřme časovač o privátní atribut:

```
uint32_t mTick_Count;
```

a o veřejnou metodu, kterou rovnou i implementujeme:

```
uint32_t Get_Tick_Count() const
{
   return mTick_Count;
}
```

Poté je třeba někde tento čítač inkrementovat – tím místem je metoda `IRQ_Callback`, kterou patřičným způsobem rozšíříme:

```
void CTimer::IRQ_Callback()
{
    Regs(hal::Timer_Reg::IRQ_Clear) = 1;

    mTick_Count++;
```

```
    if (mCallback)
        mCallback();
}
```

Nyní definujme nový stav procesu pro spánek, který dle konvencí pojmenujeme `Interruptable_Sleep`. Výčet stavů procesu tedy nyní vypadá nějak takto:

```
enum class NTask_State
{
   New,
   Runnable,
   Running,
   Blocked,
   Interruptable_Sleep,
   Zombie
};
```

V PCB nyní doplňme čas, kdy má být proces probuzen. Tento čas odpovídá vlastně počtu tiků, které musí na časovači být, aby se proces probudil. Nutno dodat, že tato hodnota bude platná pouze tehdy, je-li proces ve stavu `Interruptable_Sleep`:

```
uint32_t sleep_timer;
```

Nyní definujme výčtovou hodnotu pro službu `sleep` v `NSWI_Process_Service`:

```
// Uspi proces na dobu (ne)urcitou
// IN:  r0 = pocet tiku casovace, na kolik uspat proces
// OUT: r0 = indikator uspechu (NSWI_Result_Code), OK pokud se
    probudil po casovem useku, Fail pokud ho probudilo neco
    jineho
Sleep           = 3,
```

Obsluha je pak poměrně přímočará – nastavíme čas probuzení, změníme stav procesu a přeplánujeme:

```
case NSWI_Process_Service::Sleep:
    mCurrent_Task_Node->task->sched_counter = 1;
    mCurrent_Task_Node->task->state
          = NTask_State::Interruptable_Sleep;
    mCurrent_Task_Node->task->sleep_timer
          = sTimer.Get_Tick_Count() + r0;
```

```
    Schedule();
    break;
```

Teď už zbývá jen doplnit kód plánovače, tedy vlastně metodu `CProcess_Manager::Schedule()`, o probouzení spících procesů na začátku metody:

```
CProcess_List_Node* node = mProcess_List_Head;
while (node)
{
    if (node->task->state == NTask_State::Interruptable_Sleep)
    {
        if (node->task->sleep_timer <= sTimer.Get_Tick_Count())
            Notify_Process(node->task);
    }

    node = node->next;
}
```

3 GPIO přerušení

Abychom nějakým způsobem propojili operační systém a aplikace běžící na RPi Zero s podněty z vnějšího prostředí, implementujme nyní podporu IRQ z GPIO řadiče. To nám umožní například probouzet procesy v momentě, kdy uživatel stiskne tlačítko, přepne spínač nebo otočí zařízení tak, že sepne senzor náklonu.

Mikrokontrolér BCM2835 umí detekovat na pinu změnu, vysokou a nízkou úroveň napětí. Z dokumentace vyčteme, že se vše děje pomocí sady registrů, které doplníme do `hal/peripherals.h`:

```
GPEDS0    = 16,
GPEDS1    = 17,
GPREN0    = 19,
GPREN1    = 20,
GPFEN0    = 22,
GPFEN1    = 23,
GPHEN0    = 25,
GPHEN1    = 26,
GPLEN0    = 28,
GPLEN1    = 29,
```

Registry jsou opět ve dvojicích, aby pokryly celou sadu GPIO pinů – jednotlivé bity odpovídají každému GPIO pinu na desce. Z těchto registrů jde významově o:

- `GPESD` – v těchto registrech se objeví bit na pozici odpovídající pinu, na kterém došlo k detekci události
- `GPREN` – v tomto registru povolujeme detekci vzestupné hrany
- `GPFEN` – v tomto registru povolujeme detekci seestupné hrany
- `GPHEN` – v tomto registru povolujeme detekci vysoké úrovně napětí
- `GPLEN` – v tomto registru povolujeme detekci nízké úrovně napětí

Jak lze vypozorovat, pro povolení detekce nastavíme odpovídající pin v některém ze čtyř posledních registrů. Jakmile dojde k detekci dané události, je vyvoláno přerušení a na obslužná rutina musí vynulovat příslušný bit v registrech `GPESD`.

V tomto místě začíná být vidět důležitost jednoho z velmi důležitých principů při návrhu a implementaci jádra operačního systému. Jakmile začneme nabalovat více a více režie na obsluhu IRQ, během kterého pochopitelně není vykonáván kód uživatelských procesů, může dojít ke zpoždění při zpracování úloh, a to v obtížně detekovatelném stylu. Čím více režie je v obsluze IRQ, tím těžkopádnější systém je a tím má horší potenciál dodržet deadline úloh. Proto je obsluha typicky dělena do dvou částí – *top half* a *bottom half*. V obsluze IRQ je přesně to místo, které obvykle zpracovává *top half*, a tedy výkonnostně kritický kód. Zde by mělo dojít pouze k minimální nutné režii, obvykle ve smyslu předání IRQ do nějaké fronty v podobě aplikační struktury, vymazání příznaku přerušení a notifikace *bottom half*. Poté se pokračuje dál v programu, který IRQ přerušilo, a požadavek putuje do *bottom half*, což může být typicky nějaký proces, který je plánovaný s vyšší prioritou a běží se zvýšeným oprávněním. Na architektuře ARM jde o proces, jehož kód běží v systémovém režimu (narozdíl od uživatelských programů, které běží v user režimu).

Pro představu – teď ještě není nutné nic takového odkládat, jelikož jde jen o jednoduché „prostupné“ schéma, kdy jen na základě příchozího IRQ notifikujeme konkrétní proces. Jakmile bychom ale implementovali například WiFi driver a v jádře měli podporu transportního protokolu TCP, jehož režie (která jde vždy mimo uživatelský proces a zůstává v jádře) je pochopitelně řádově větší (porty, spolehlivost, udržení spojení, ...), zjistili bychom, že obsluha IRQ z WiFi koprocesoru trvá déle, než v současnosti použitá jednotka časového kvanta, kterou určuje časovač. V tento moment je na světě výrazný problém.

K rozdělení *top* a *bottom half* se dostaneme nejspíše během dalších cvičení. Pro teď implementujme jen to nejnutnější, což pro nás bude detekce IRQ na GPIO pinu a notifikace koncového procesu.

3.1 Úprava driveru

Definujme nyní výčtový typ druhu události:

```
enum class NGPIO_Interrupt_Type
{
   Rising_Edge,
   Falling_Edge,
   High,
   Low,
};
```

A nyní postupně rozšiřujme GPIO driver o potřebné metody. Jako první vytvořme metody pro zjištění pozice registrů a indexu pinů pro event detection:

```
bool CGPIO_Handler::Get_GPEDS_Location(uint32_t pin, uint32_t&
    reg, uint32_t& bit_idx) const
{
   if (pin > hal::GPIO_Pin_Count)
   return false;

   reg = static_cast<uint32_t>((pin < 32) ? hal::GPIO_Reg::GPEDS0
     : hal::GPIO_Reg::GPEDS1);
   bit_idx = pin % 32;

   return true;
}

bool CGPIO_Handler::Get_GP_IRQ_Detect_Location(uint32_t pin,
    NGPIO_Interrupt_Type type, uint32_t& reg, uint32_t& bit_idx)
    const
{
   if (pin > hal::GPIO_Pin_Count)
      return false;

   bit_idx = pin % 32;

   switch (type)
   {
      case NGPIO_Interrupt_Type::Rising_Edge:
         reg = static_cast<uint32_t>((pin < 32) ?
                hal::GPIO_Reg::GPREN0 : hal::GPIO_Reg::GPREN1);
         break;
      case NGPIO_Interrupt_Type::Falling_Edge:
         reg = static_cast<uint32_t>((pin < 32) ?
                hal::GPIO_Reg::GPFEN0 : hal::GPIO_Reg::GPFEN1);
```

```
         break;
      case NGPIO_Interrupt_Type::High:
         reg = static_cast<uint32_t>((pin < 32) ?
                hal::GPIO_Reg::GPHEN0 : hal::GPIO_Reg::GPHEN1);
         break;
      case NGPIO_Interrupt_Type::Low:
         reg = static_cast<uint32_t>((pin < 32) ?
                hal::GPIO_Reg::GPLEN0 : hal::GPIO_Reg::GPLEN1);
         break;
      default:
         return false;
   }

   return true;
}
```

Nyní můžeme implementovat metody pro povolení (zakázání) detekce události pomocí daného výčtového typu:

```
void CGPIO_Handler::Enable_Event_Detect(uint32_t pin,
    NGPIO_Interrupt_Type type)
{
   uint32_t reg, bit;
   if (!Get_GP_IRQ_Detect_Location(pin, type, reg, bit))
      return;

   mGPIO[reg] = (1 << bit);

   // NOTE: pro ted takto, do budoucna samozrejme vyresme cisteji
   sInterruptCtl.Enable_IRQ(hal::IRQ_Source::GPIO_0);
   sInterruptCtl.Enable_IRQ(hal::IRQ_Source::GPIO_1);
   sInterruptCtl.Enable_IRQ(hal::IRQ_Source::GPIO_2);
   sInterruptCtl.Enable_IRQ(hal::IRQ_Source::GPIO_3);
}

void CGPIO_Handler::Disable_Event_Detect(uint32_t pin,
    NGPIO_Interrupt_Type type)
{
   uint32_t reg, bit;
   if (!Get_GP_IRQ_Detect_Location(pin, type, reg, bit))
      return;

   uint32_t val = mGPIO[reg];
   val &= ~(1 << bit);
   mGPIO[reg] = val;
```

```
}
```

Jakmile dojde k přerušení, je nastaven příslušný bit v `GPEDS` registru. Získejme jeho pořadí pomocí metody:

```
uint32_t CGPIO_Handler::Get_Detected_Event_Pin() const
{
   uint32_t reg, bit;
   for (uint32_t i = 0; i < hal::GPIO_Pin_Count; i++)
   {
      if (!Get_GPEDS_Location(i, reg, bit))
         return Invalid_Pin;

      if ((mGPIO[reg] >> bit) & 0x1)
         return i;
   }

   return Invalid_Pin;
}
```

Nutno dodat, že tato metoda nic nemaže – příznak v registru zůstává, dokud se ho sami nerozhodneme odstranit. Implementujme si pro to další metodu, která nastavením bitu v příslušném `GPEDS` registru vymaže příznak čekajícího přerušení:

```
void CGPIO_Handler::Clear_Detected_Event(uint32_t pin)
{
   uint32_t reg, bit;
   if (!Get_GPEDS_Location(pin, reg, bit))
      return;

   mGPIO[reg] = 1 << bit;
}
```

Nyní potřebujeme dva dílčí mechanismy pro úspěšné předávání událostí z GPIO driveru do uživatelských procesů. Prvním z nich je vložení záznamu čekajícího procesu do interní fronty. To ale trochu obejdeme z druhé strany – do GPIO driveru nebude vcházet záznam ani identifikátor procesu, ale rovnou třída souboru, která sama o sobě zprostředkovává potřebné rutiny procesu skrze své rozhraní. Druhou částí je obsluha IRQ, která příslušné procesy probouzí, respektive notifikuje soubory z fronty.

Vytvořme proto záznam fronty, klidně jako privátní strukturu driveru, a hned instancujme frontu v driveru:

```
struct TWaiting_File
{
   IFile* file;
   uint32_t pin_idx;
   TWaiting_File* prev;
   TWaiting_File* next;
};

TWaiting_File* mWaiting_Files;
```

Teď můžeme implementovat řazení souborů do fronty k notifikování:

```
void CGPIO_Handler::Wait_For_Event(IFile* file, uint32_t pin)
{
   // TODO: zamknout zamek

   TWaiting_File* wf = new TWaiting_File;
   wf->file = file;
   wf->pin_idx = pin;
   wf->prev = nullptr;
   wf->next = mWaiting_Files;

   mWaiting_Files = wf;

   // TODO: odemknout zamek
}
```

A rovnou i metodu, která bude zpracovávat příchozí IRQ:

```
void CGPIO_Handler::Handle_IRQ()
{
   TWaiting_File* wf, *tmpwf;

   uint32_t reg, bit, pin;
   for (pin = 0; pin < hal::GPIO_Pin_Count; pin++) {
      if (!Get_GPEDS_Location(pin, reg, bit))
         continue;

      if ((mGPIO[reg] >> bit) & 0x1) {
         // TODO: zamknout zamek

         wf = mWaiting_Files;
         while (wf != nullptr) {
            if (wf->pin_idx == pin) {
```

```
                wf->file->Notify(NotifyAll);

                if (wf->prev)
                   wf->prev->next = wf->next;
                if (wf->next)
                   wf->next->prev = wf->prev;

                tmpwf = wf;

                if (mWaiting_Files == wf)
                   mWaiting_Files = wf->next;
                wf = wf->next;
                delete tmpwf;
             }
             else
                wf = wf->next;
          }

          // TODO: odemknout zamek

          Clear_Detected_Event(pin);
       }
    }
}
```

Už zbývá jen nechat volat tuto obsluhu z generické obsluhy – proto do `_internal_interrupt_handler` vložme volání `sGPIO.Handle_IRQ();`.

3.2 Úprava GPIO FS driveru

Jelikož metoda `Notify` nad objektem typu `IFile` funguje genericky, stačí nám jen správně uspat proces a zařadit ho do fronty. To v řeči rozhraní `IFile` znamená implementovat metodu `Wait`:

```
virtual bool Wait(uint32_t count) override
{
   Wait_Enqueue_Current();
   sGPIO.Wait_For_Event(this, mPinNo);

   sProcessMgr.Block_Current_Process();
   return true;
}
```

Zde nejprve vložíme současný proces do interní fronty `IFile`, předáme sami sebe do GPIO driveru jako notifikovatelný soubor nad daným GPIO pinem a následně zablokujeme současný proces. Nutno dodat, že v tomto místě se současný proces zastaví a po svém probuzení zde bude pokračovat – dojde zde k přeplánování na proces jiný.

3.3 Rozšíření IOCtl

Blokovat se nad souborem a probouzet se již umíme. Poslední částí, kterou musí jádro zpřístupnit, je způsob, jakým může uživatelský proces povolit detekci událostí, aby to vůbec mělo smysl.

Již jsme v minulosti definovali systémové volání `ioctl`, které dovolilo měnit parametry souboru specifickým způsobem pro jeho druh. Do tohoto volání doplníme takovou parametrizaci, která dovolí zapnout a vypnout detekci události na souboru, ať už je jakéhokoliv typu. Implementaci demonstrujeme na GPIO souborech, jelikož jde o jediné soubory, které momentálně detekci události podporují.

Do výčtového typu `NIOCtl_Operation` (v `swi.h`) doplňme dvě konstanty:

```
Enable_Event_Detection  = 2,
Disable_Event_Detection = 3,
```

Díky hotové implementaci v driveru pak lze jen dodefinovat v GPIO FS driveru v implementaci souboru metodu `IOCtl`:

```
virtual bool IOCtl(NIOCtl_Operation op, void* ctlptr) override
{
   NGPIO_Interrupt_Type evtype =
         *reinterpret_cast<NGPIO_Interrupt_Type*>(ctlptr);

   switch (op)
   {
      case NIOCtl_Operation::Enable_Event_Detection:
         sGPIO.Enable_Event_Detect(mPinNo, evtype);
         return true;
      case NIOCtl_Operation::Disable_Event_Detection:
         sGPIO.Disable_Event_Detect(mPinNo, evtype);
         return true;
   }

   return false;
}
```

V tuto chvíli je to vše, co je potřeba k úspěšné detekci změny stavu na GPIO pinu a jeho předání do uživatelského procesu

3.4 Polling vs. IRQ

Kdybychom neměli mechanismus přerušení, museli bychom v každém momentě provádění uživatelského procesu periodicky zjišťovat hodnotu na daném pinu, abychom změnu detekovali. To se samozřejmě podepíše jednak na celkovém výkonu systému, kdy jsou zbytečně „spáleny“ spousty cyklů jen na čtení stavu pinu, a jednak jde pochopitelně také o frekvenci čtení.

Kdyby totiž náhodou došlo k této události v momentě, kdy daný proces provádějící polling není vůbec naplánovaný (je třeba ve stavu `Runnable`, tedy běží jiný proces), a pin by se stihl vrátit do původního stavu, událost bychom propásli. To je ovšem pohled uživatelských procesů.

Pohled jádra je diametrálně odlišný – jádro může provádět polling cíleně místo IRQ proto, aby nepřidávalo režii za situace, kdy očekává, že se daná událost zpracuje v nejbližší době. Tohle je typické například pro řadiče fyzických úložišť nebo jiných externích pamětí, kdy odezva na požadavek přijde typicky do několika taktů procesoru. Není třeba nic uspávat, jelikož by se tím odezva jádra na požadavek procesu naopak řádově zvýšila.

4 Synchronizace

Jelikož umíme uspávat a notifikovat procesy, je nejvyšší čas implementovat základní synchronizační primitiva. Nutno dodat, že v této sekci se nebudeme zaobírat maximální efektivitou implementace, jelikož ta předpokládá systémové prvky, které jednoduše v našem systému nemáme. Do reálného systému je pochopitelně žádoucí veškerou synchronizaci procesů maximálně zefektivnit, jelikož jde o výrazný faktor v celkové výkonnosti.

Rovněž navrhněme tyto synchronizační prvky tak, aby byly implicitně sdílené mezi procesy. V našem systému teď ani nedovolujeme vytvářet vlákna nebo korutiny, a tak je bezpředmětné podporovat tyto prvky jen v rámci jednoho procesu. Všechny budou tedy sdílené mezi procesy, které si budou odkazy na ně předávat prostřednictvím souborového systému.

Budeme implementovat:

- spinlock – ten bude používaný jen v jádře (nebude tedy v souborovém systému)
- mutex – cesta `SYS:mtx/<nazev>`
- semafor – cesta `SYS:sem/<nazev>#<pocet>`, kde `<pocet>` je počet zdrojů (popř. místo počtu lze dodat `?`, pokud by měl semafor vytvářet jiný proces)
- podmínková proměnná – cesta `SYS:cv/<nazev>`
- pojmenovaná roura – cesta `SYS:pipe/<nazev>#<pocet>`, kde `<pocet>` je velikost roury (popř. opět lze dodat `?`)

V souborovém systému budou tyto prostředky vytvářené vždy s prvním otevřením, a odstraňované s posledním zavřením. Každý prostředek bude tedy udržovat čítač referencí.

4.1 Spinlock

Spinlock je druh zámku, na který je v případě, že není k dispozici, nutné aktivně čekat. Aktivní čekání zde spočívá v cyklickém dotazování na stav zámku. Jde o základní způsob synchronizace v jádře operačního systému. Samozřejmě ale vyvstává otázka, kdy je spinlock vhodné použít. Spinlock je rozhodně dobré použít za předpokladu, kdy je reálná šance, že pokud zámek není k dispozici, bude uvolněn během několika málo cyklů. To znamená, že musí jiný proces zámek uvolnit, zatímco jiný aktivně čeká. Z toho plyne, že hlavní význam spinlocků je v multiprocesorovém jádře.

Jelikož vyvíjíme uniprocesorové jádro, spinlock takový význam mít nebude. My ho ale přesto implementujeme, abychom demonstrovali princip spinlocku na architektuře ARM a na použití v různých místech systému.

O problematice spinlocku např. v jádře GNU/Linux dále pojednává například `https://www.kernel.org/doc/Documentation/locking/spinlocks.txt`, případně `https://0xax.gitbooks.io/linux-insides/content/SyncPrim/linux-sync-1.html`.

Implementujme si tedy jednoduchý spinlock. Vytvořme hlavičkový soubor `spinlock.h` a implementaci `spinlock.s`. V hlavičce definujme základní rozhraní spinlocku:

```
using spinlock_t = int;

constexpr uint32_t Lock_Unlocked = 0;
constexpr uint32_t Lock_Locked   = 1;

extern "C" void spinlock_init(spinlock_t* lock);
extern "C" uint32_t spinlock_try_lock(spinlock_t* lock);
extern "C" void spinlock_unlock(spinlock_t* lock);

inline void spinlock_lock(spinlock_t* lock)
{
   while (!spinlock_try_lock(lock))
   ;
}
```

Klasicky rozhraní obsahuje inicializaci, funkci pro pokus o zamčení, a pro odemčení. Implementace pak vypadá pro inicializaci takto:

```
.equ Lock_Locked,   1
.equ Lock_Unlocked, 0

spinlock_init:
   mov r12, #Lock_Unlocked
   str r12, [r0]
   bx lr
```

Zamčení pak musí být provedeno exkluzivní load/store instrukcí, která zajistí, že dojde k „prohození“ registru a paměti za předpokladu, že úvodní čtení z paměti přečetlo hodnotu „uzamčeno“. K tomu má ARM instrukce exkluzivního load () a store (). Zamčení pak může vypadat třeba takto:

```
spinlock_try_lock
   mov r1, #Lock_Locked
   ldrex r2, [r0]
   cmp r2, #Lock_Unlocked
   strexeq r3, r1, [r0]
   cmpeq r3, #Lock_Unlocked
   mov r0, r2
   bx lr
```

Výše uvedený kód načte do `r2` aktuální hodnotu zámku, a pokud byl odemčený, pokusí se zapsat hodnotu „uzamčeno“. Následně porovná starou hodnotu s hodnotou „odemčeno“ a vrací se zpět.

Kód pro odemčení je pak už jednoduchý – předpokládá, že zámek máme a vrací zpět hodnotu „odemčeno“:

```
spinlock_unlock:
   mov r12, #Lock_Unlocked
   str r12, [r0]
   bx lr
```

Tato část bude ještě doplněna...

4.2 Resource manager

Tato část bude doplněna...

4.3 Mutex

Mutex bude v našem systému jen speciální druh souboru – jediný rozdíl bude v konkrétní implementaci rozhraní. Nově implementované metody `Wait` a `Notify` budou vlastně ve výsledku sloužit k uzamčení (`Wait`) a odemčení `Notify`.

Implementujme tedy třídu mutexu `CMutex`, kde implementujeme základ a rozhraní `IFile`:

```
class CMutex : public IFile
{
   private:
```

```
      unsigned int mHolder_PID = 0;

      spinlock_t mLock_State = Lock_Unlocked;

   public:
      CMutex();

      bool Lock();

      bool Try_Lock();

      bool Unlock();

      unsigned int Get_Holder_PID() const { return mHolder_PID; }

      virtual uint32_t Read(char* buffer, uint32_t num) override
      {
         return 0;
      }
      virtual uint32_t Write(const char* buffer, uint32_t num)
            override {
         return 0;
      }

      virtual bool Close() override {
         return true;
      }
      virtual bool IOCtl(NIOCtl_Operation dir, void* ctlptr)
            override {
         return false;
      }

      virtual bool Wait(uint32_t count) override {
         return Lock();
      }
      virtual uint32_t Notify(uint32_t count) {
         return Unlock();
      }
};
```

Jak je vidět, mutex má jak metodu `Lock`, tak `Wait` (jak `Unlock`, tak `Notify`) – jedno rozhraní zveřejníme pro jádro, a jedno pro uživatelské procesy.

Taktéž je vidět, že jádrem mutexu je vlastně spinlock. V implementaci je vidět, že to, co se tváří jako spinlock je zde použito jen jako zámek, který se mutex pokusí zabrat. Když se to nepovede (mutex vlastní někdo jiný), uspí se

a předá řízení jinému procesu:

```
bool CMutex::Lock()
{
   auto* cur = sProcessMgr.Get_Current_Process();
   const unsigned int cpid = cur->pid;

   if (mHolder_PID == cpid)
      return false;

   while (spinlock_try_lock(&mLock_State) == Lock_Locked)
   {
      Wait_Enqueue_Current();
      sProcessMgr.Block_Current_Process();
   }

   mHolder_PID = cpid;

   return true;
}
```

Za zmínku stojí ještě metoda pro odemčení, která odemkne zámek a notifikuje jeden z čekajících procesů:

```
bool CMutex::Unlock()
{
   auto* cur = sProcessMgr.Get_Current_Process();
   const unsigned int cpid = cur->pid;

   if (mHolder_PID != cpid || mLock_State != Lock_Locked)
      return false;

   mHolder_PID = 0;
   spinlock_unlock(&mLock_State);

   return IFile::Notify(1);
}
```

Pochopitelně by bylo o něco lepší, kdyby byl zde vybrán konkrétní proces, a zámek mu byl bez odemčení předán. Ponechme to ale teď takto, abychom použili generických prostředků, které jsme již implementovali.

4.4 Semafor

Tato část bude doplněna...

4.5 Podmínková proměnná

Tato část bude doplněna...

4.6 Pojmenovaná roura

Tato část bude doplněna...

5 Earliest Deadline First plánovač

Earliest Deadline First (EDF) plánovač plánuje vždy takový proces, který má nejdřívější deadline. Ke správné funkci tedy musíme mít způsob, jakým dodat plánovači informaci o deadline. Vzhledem k modelu tasků, který jsme zvolili, budou všechny procesy běžet po celou dobu běhu systému. Každý proces bude čekat (syscall `wait`) na systémový prostředek, a nebo bude uspaný na daný čas (syscall `sleep`), než bude znovu plánován. Tímto oddělíme periodické a aperiodické tasky.

Deadline bude nastavena vždy po probuzení. Musíme tedy jednak specifikovat tu danou deadline v systémovém volání, a jednak donutit systém tuto deadline po probuzení použít. Pak stačí jen implementovat plánovač a EDF by mělo fungovat.

5.1 Deadline

Do PCB proto doplňme dvě položky:

```
uint32_t deadline;
uint32_t notified_deadline;
```

Navíc ještě definujme dvě pomocné hodnoty:

```
constexpr uint32_t Indefinite
                       = static_cast<uint32_t>(-1);
constexpr uint32_t Deadline_Unchanged
                       = static_cast<uint32_t>(-2);
```

Konstanta `Indefinite` označuje v kontextu deadline takový proces, který deadline vůbec nemá. Konstanta `Deadline_Unchanged` dává řídicímu kódu najevo, že nemá s deadline vůbec hýbat.

Rozhodně je třeba v inicializaci procesu nastavovat `deadline` na `Indefinite` – na začátku žádný proces žádnou nemá. Rozšiřme metodu `Notify_Process` tak, aby po notifikaci (probuzení) deadline nastavovala:

```
if (proc->notified_deadline != Deadline_Unchanged)
{
   proc->deadline = sTimer.Get_Tick_Count()
                    + proc->notified_deadline;
}
```

Nyní ještě potřebujeme tuto „odloženou“ deadline nastavovat. Rozšiřme proto systémové volání `wait` a `sleep` o další parametr – v registru `r2` (`wait`) a `r1` (`sleep`) vyžadujme relativní čas, do kterého má úloha být zpracována po probuzení procesu (tedy po přijetí podnětu). Tak jsme schopni docílit efektu, kdy specifikujeme deadline ještě před uspáním, a není tedy nutné řešit dodatečné nastavování, až když k události dojde. Takhle je deadline nastavena současně s momentem, kdy dojde k probuzení.

Důležitá je ale i samotná implementace – `wait` totiž může, ale nemusí blokovat, a proto deadline nastavujeme také hned před voláním implementace `Wait` v `IFile`. Rovněž po návratu z `Wait` nesmíme zapomenout tuto připravenou deadline zase smazat:

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

   mCurrent_Task_Node->task->notified_deadline = r2;

   if (r2 != Deadline_Unchanged)
      mCurrent_Task_Node->task->deadline = r2;

   target.r0 = mCurrent_Task_Node->task->opened_files[r0]->Wait(
    r1);

   mCurrent_Task_Node->task->notified_deadline =
    Deadline_Unchanged;
   break;
}
```

Stejě tak rozšiřme obsluhu `sleep`:

```
mCurrent_Task_Node->task->notified_deadline = r1;
```

5.2 Plánování

Abychom mohli měnit nebo různě řetězit strategie plánování, definujme ve správci procesů ukazatel na funkci, která bude plnit funkci plánování:

```
CProcess_List_Node* (CProcess_Manager::*mSchedule_Fnc)() =
    nullptr;
```

Rovnou vytvořme ve správci i definice plánovacích funkcí pro současný round-robin a pro nový EDF:

```
CProcess_List_Node* Schedule_RR();
CProcess_List_Node* Schedule_EDF();
```

Samotná metoda `Schedule` se tedy zjednoduší:

```
void CProcess_Manager::Schedule()
{
   CProcess_List_Node* node = mProcess_List_Head;
   while (node)
   {
      if (node->task->state == NTask_State::Interruptable_Sleep)
      {
         if (node->task->sleep_timer != Indefinite &&
             node->task->sleep_timer <= sTimer.Get_Tick_Count())
            Notify_Process(node->task);
      }
      node = node->next;
   }

   CProcess_List_Node* next = (this->*mSchedule_Fnc)();
   if (!next)
      return;

   if (next == mCurrent_Task_Node)
      return;

   Switch_To(next);
}
```

Round-robin plánovač jen přesuňme do metody `Schedule_RR()`. Zajímavější je pro nás EDF plánovač, který plánuje ty `Runnable` procesy, které mají nejkratší deadline – plánovač tedy vždy vybere ze seznamu procesů ten s nejkratší deadline a naplánuje ho:

```
CProcess_List_Node* CProcess_Manager::Schedule_EDF()
{
   CProcess_List_Node* next = nullptr;

   CProcess_List_Node* itr = mProcess_List_Head;
   while (itr)
   {
      if (itr->task->state != NTask_State::New &&
         itr->task->state != NTask_State::Runnable &&
         itr->task->state != NTask_State::Running)
      {
         itr = itr->next;
         continue;
      }

      if (!next)
         next = itr;
      else if (itr->task->deadline != Indefinite)
      {
         if (next->task->deadline > itr->task->deadline)
            next = itr;
      }

      itr = itr->next;
   }

   if (next && next->task->deadline == Indefinite)
      return Schedule_RR();

   return next;
}
```

Nutno dodat, že pokud žádný (plánovatelný) proces nemá nastavenou deadline, degraduje plánovací politika zpět na round-robin.

Nakonec zbývá nastavit ve správci procesů v konstruktoru plánovací funkci a vše by mělo začít fungovat:

```
mSchedule_Fnc = &CProcess_Manager::Schedule_EDF;
```

Co je ovšem také nutné brát v potaz je to, kdy vlastně plánovač spouštět. Doteď jsme ho spouštěli vždy při tiknutí časovače nebo explicitně při blokování procesu. EDF bychom měli rovněž spouštět periodicky (kvůli periodickým taskům a možnosti, že se nějaký proces probudí), při blokování procesu, ale nově i při

notifikování procesu – probouzený proces může mít kratší deadline (a tedy vyšší prioritu).

Dále je pochopitelně velice dobře možné, že si jeden task s nižší prioritou bude držet systémové zdroje, které vyžaduje task s prioritou vyšší. Pak by mohlo dojít k nesplnění deadline jen proto, že se plánovač neuměl zařídit tak, aby se procesy vystřídaly. Z tohoto důvodu je zaváděn princip *inverze priorit*. Ten dovolí dočasně tasku s nízkou prioritou převzít prioritu tasku jiného (s prioritou vyšší), aby měl šanci zdroj předat dříve, než dojde k nedodržení deadline. Tento problém pochopitelně není vidět, jsou-li v plánovači jen dva procesy – tam task s nižší prioritou poběží vždy, když ten druhý nemůže být plánován. Problém zviditelní až přítomnost procesu dalšího, který má například prioritu přesně mezi těmito dvěma kritickými procesy (tedy má deadline někde mezi). Problémem inverze priorit se ale pro teď zaobírat nebudeme – možná v dalších cvičeních.

6 Power management

Klíčovou vlastností embedded zařízení (tedy typických hostitelských zařízení systémů reálného času) je i schopnost do značné míry šetřit elektrickou energii. Toho pochopitelně musí docílit hlavně sám programátor jádra systému, jelikož musí zajistit, aby se včas všechny součásti systému vypínaly nebo se přepínaly do režimu s nízkou spotřebou.

Kromě periferií a jejich uspávání (což je tedy záležitost driverů) jde jednoznačně o pasivitu systému v momentě, kdy se nic neděje. V tomto případě by pouze *idle* task donekonečna cyklil v nekonečné smyčce a bral by tedy čas procesoru na neužitečnou práci, což pochopitelně znamená plný odběr elektrické energie jádrem procesoru.

ARM ale obsahuje dvě důležité power-management instrukce: `WFI` (*wait for interrupt*) a `WFE` (*wait for event*). Ty dovolí jádro procesoru uspat do momentu, než nastane přerušení (`WFI`) nebo událost (`WFE`). Událost je v tomto kontextu jakousi nadmnožinou, takže zahrnuje i přerušení – `WFE` je tedy „obecnější“, než `WFI`. Nutno dodat, že specifikace ARM neukládá povinnost tyto instrukce podporovat, ale spousty moderních implementací je obsahuje.

Vzhledem k tomu, že implementujeme uniprocesorový kernel se soustřeďme hlavně na `WFI`. To uspí procesorové jádro do doby, než přijde nějaké přerušení (IRQ, FIQ, data abort a jiné). Jak jsme naznačili, probouzení procesu je vždy podmíněno nějakým přerušením, a to buď přerušením externí periferie (GPIO, komunikační sběrnice, ...) (uspání na `wait`), nebo přerušením časovače (uspání na `sleep`).

Kdybychom dělali multiprocesorový (SMP) kernel, určitě bychom chtěli implementovat `WFE`, a to pro všechna procesorová jádra – například naplánovat idle task s afinitou pro každé jádro. `WFE` je zde potřebným nástrojem proto, že to navíc umožňuje probouzet procesor na žádost jiného procesoru, který signalizuje událost instrukcí `SEV`.

Můžeme tedy směle modifikovat *idle* task tak, aby pokud není v systému plánovatelný jiný proces, aby spustil instrukci `WFI`. Pokud ano, předá pouze

zbytek časového kvanta jinému procesu systémovým voláním `yield`:

```
while (true)
{
   if (get_active_process_count() == 1)
      asm volatile("wfi");

   sched_yield();
}
```

Volání `get_active_process_count` snadno implementujeme jako systémové volání (ponecháno na fantazii čtenáře), které vrátí počet procesů, které jsou momentálně ve stavu `New`, `Runnable` nebo `Running`. Logicky pak vyplývá, že pokud toto volání vrátí číslo 1, je *idle* task jediným plánovatelným a lze tedy bezpečně procesor uspat instrukcí `WFI`. V každém případě (i kdyby po probuzení) task předá časové kvantum dalšímu tasku, pokud nějaký je.

Tímto jsme schopni docílit relativně snadno výrazné úspory – u systémů reálného času bývá typické, že ve spoustě praktických aplikací většinu času spí, a tedy by měly šetřit elektrickou energii. Obzvláště pak pokud jsou napájeny z baterie nebo jiného omezeného zdroje energie.

Prakticky je třeba ještě doladit i frekvenci tikání časovače, aby vůbec mělo smysl procesor uspávat, když by mělo za zlomek času následovat přerušení časovače (který třeba vůbec nemá v úmyslu jakýkoliv proces probudit). Toto nastavení bývá pak silně specifické pro konkrétní aplikace.

7 Úkol za body

Pokuste se (stačí slovně) navrhnout, jak by takový systém mohl implementovat detekci uvíznutí. Pod pojmem detekce uvíznutí si pro potřeby tohoto úkolu za body představte v podstatě jakkoliv nákladný algoritmus, který je schopen detekovat, že se procesy dostaly do stavu, ve kterém se navzájem donekonečna blokují. Nevymýšlejte kompletní detekci, to je složitý problém, který nemusí mít řešení. Stačí, když se pokusíte popsat algoritmus, kterým lze detekovat alespoň jeden z nějakých typických stavů (např. procesy v grafu alokace zdrojů vytvořily jednoduchý cyklus).

Připomeňme pro referenci Coffmannovy podmínky uvíznutí:

1. vzájemné vyloučení – náš OS umí zajistit
2. hold and wait – náš OS nezakazuje žádat další prostředky
3. neodnímatelnost – náš OS ani neumí odejmout zdroj, aniž by porušil konzistenci
4. cyklické čekání – je perfektně možné dostat se do stavu, kdy v grafu alokace zdrojů vznikne cyklus

Rovněž připomínám, že nejde o řešení uvíznutí, ale pouze jeho detekci. Pokuste se tedy navrhnout základní algoritmus detekce uvíznutí pro náš operační systém.

Odpovězte tedy na 2 základní otázky: *kdy? jak?*

Za tento úkol můžete získat až 2 body.