KIV/OS - cvičení č. 5

Martin Úbl

1. srpna 2021

1 Obsah cvičení

- alokátor paměti
- kernelová halda
- preemptivní round-robin plánovač

2 Alokátor paměti

Alokace paměti je obecně poměrně složitý problém, pokud to chceme udělat správně a co nejefektivněji. My implementujeme v rámci cvičení takový alokátor, který bude pouze velmi primitivní. Rozhraní však bude mít finální a lepší implementace pak bude ponechána na další cvičení.

Nyní náš operační systém nezná např. ani stránkování – to přibyde až v některém z dalších cvičení. Alokátor proto moc optimalizovat nebudeme, jelikož schéma alokace pak bude určitě odlišné. Mohli bychom ale už trochu počítat s tím, že budeme chtít alokovat stránky o nějaké konkrétní velikosti. Rovněž rovnou počítejme s tím, že oblast, ve které chceme alokovat, bude nějak zdola a shora omezena.

Definujme si proto soubor `memmap.h`, kde si tyto meze stanovíme. Oblast začátku můžeme zvolit v podstatě libovolně tak, aby se nepřekrývala s kódem (a daty) jádra. Pro teď si zvolme nějaký počátek ručně, v budoucnu však můžeme například použít již známý trik – v linker skriptu definovat zarážku za poslední sekcí a tu si vyzvednout. Pak je třeba ji zarovnat na nejbližší vyšší násobek velikosti stránky a lze ji použít.

Horní mez stanovme dle možností architektury a desky. V manuálu se dočteme, že nejbližší horní hranicí je memory-mapped IO region. Pak můžeme horní mez nastavit prostě na `Peripheral_Base`. Pozor ale – naše zařízení zdaleka tolik fyzické paměti nemá. Řešením by mohlo být například vyzvednutí skutečného rozsahu paměti pomocí mailbox mechanismu. To ale není obsahem tohoto cvičení.

Obsah souboru `memmap.h` pak může vypadat třeba takto:

```
namespace mem
{
  constexpr uint32_t LowMemory = 0x20000;
  constexpr uint32_t HighMemory = Peripheral_Base;
  constexpr uint32_t PageSize = 0x4000;
  constexpr uint32_t PagingMemorySize =
                          HighMemory - LowMemory;
  constexpr uint32_t PageCount =
                       PagingMemorySize / PageSize;
}
```

Pokud to ještě v tento moment není zřejmé, operujeme v módu, kdy je každý kus paměti přímo přístupný a jeho adresa není nijak překládána. Pokud se tedy jádro v jakýkoliv moment rozhodne, že chce něco zapsat do paměti na adrese `0x00123456`, tak se to prostě s největší pravděpodobností povede.

Implementací alokátoru vlastně všechnu dostupnou paměť „přidělíme“ právě jádru, a alokátor bude tuto paměť spravovat. Měl by podporovat jak alokaci, tak i dealokaci bloku paměti. Z podstaty věci ale samotný alokátor neví, komu paměť přiděluje – to pak je otázkou příslušného „správce“. Tím je třeba správce haldy jádra, správce procesů a tak dále. Tento správce zodpovídá jak za „úměrnou“ alokaci, tak za správnou a úplnou dealokaci. Rovněž může dále vnitřně paměť rozdělovat na menší kousky – to typicky budeme chtít jak u haldy jádra, tak i u procesů. K tomu ale zase později.

Jednou z nejjednodušších implementací alokátorů stránek (resp. později rámců) je bitová mapa. V té každý bit reprezentuje jednu stránku (rámec) paměti, přičemž hodnota „vypnuto“ znamená, že je blok volný a „zapnuto“, že byl někomu přidělen. Jelikož víme, kolik stránek můžeme maximálně alokovat (`PageCount`), tak také víme, jak velkou bitovou mapu budeme potřebovat.

Uveďme příklad. Za předpokladu, že je paměť velká 512 MiB ($512 * 1024 * 1024$ B) a stránka (rámec) má velikost `0x4000` (16 kiB, $16 * 1024$ B), pak tato bitová mapa musí čítat 32768 bitů (4096 B, tedy 4 kiB). Tento výpočet lze provést v čase překladu přímo z konstant. Obětování 4 kiB z paměti pro bitmapu, která reprezentuje 512 MiB paměťový prostor není zase tolik. Jde to ale samozřejmě trochu lépe a efektivněji – o tom více pojednávají přednášky KIV/OS nebo KIV/ZOS. Pro jednoduchost tady na cvičení implementujme bitovou mapu s algoritmem typu *first-fit*, tedy $O(n)$ alokátorem, který hledá první volný blok od začátku alokovatelné paměti.

Mějme tedy třídu `CPage_Manager`, která bude naším alokátorem. Ta bude obsahovat metodu `Alloc_Page()`, která bude alokovat novou stránku a bude vracet adresu prvního bajtu (nebo 0, pokud žádná stránka není volná). Dále bude obsahovat metodu `Free_Page(uint32_t addr)`, která bude přejímat adresu prvního bajtu stránky k uvolnění jako parametr a stránku bude ve své implementaci uvolňovat.

Implementujeme bitmapový *first-fit*, a tedy potřebujeme mít alokovanou bitmapu. Ve třídě proto vytvořme atribut, který bude představovat bitovou

mapu alokovatelné paměti `uint8_t mPage_Bitmap[mem::PageCount / 8];`.

Samotná implementace pak musí obsahovat inicializaci (zde nejlépe asi konstruktorem), ve které označíme všechny bloky za volné:

```
CPage_Manager::CPage_Manager()
{
  for (int i = 0; i < sizeof(mPage_Bitmap); i++)
    mPage_Bitmap[i] = 0;
}
```

Alokace pak probíhá snadno – nalezením prvního oktetu stránek, který není celý alokovaný (abychom to trochu urychlili), v něm nalezení volné stránky a následným označením a vrácením:

```
uint32_t CPage_Manager::Alloc_Page()
{
  uint32_t i, j;

  for (i = 0; i < mem::PageCount; i++)
  {
    if (mPage_Bitmap[i] != 0xFF)
    {
      for (j = 0; j < 8; j++)
      {
        if ((mPage_Bitmap[i] & (1 << j)) == 0)
        {
          const uint32_t page_idx = i*8 + j;
          mPage_Bitmap[page_idx / 8] |= 1 << (page_idx % 8);
          return mem::LowMemory + page_idx * mem::PageSize;
        }
      }
    }
  }

  return 0;
}
```

Uvolnění je pak již snadné – stačí příslušný bit vynulovat:

```
void CPage_Manager::Free_Page(uint32_t addr)
{
  const uint32_t page_idx = addr / mem::PageSize;
  mPage_Bitmap[page_idx / 8] &= ~(1 << (page_idx % 8));
}
```

V systémech, kde je nutné dbát na bezpečnost (typicky spíše interaktivní systémy) je možné při dealokaci ještě systémově přemazat obsah dealokované paměti (nějakým vzorem nebo náhodnými daty) – pokud by k tomu nedošlo, ve stránce byla nějaká citlivá data (hesla, šifrovací klíče, ...) a stránku si pak alokoval nějaký jiný proces, k datům by pak měl přístup.