# Mapování souborů do paměti v Linuxu

Cvičení VII

## 1 Práce se soubory

Mapování souborů do paměti je praktický nástroj, který nám umožňuje pracovat s daty v souborech stejným způsobem jako s daty v paměti. U tohoto způsobu práce nejsme omezeni na operace typu read/write, ale můžeme využívat všechny operace pro práci s pamětí, které nám programovací jazyk nabízí. Dále nám mapování souborů do paměti umožňuje pracovat s velkými soubory, přičemž o tom, které části souboru jsou v daný moment skutečně v paměti, rozhoduje správa virtuální paměti, která transparentně řeší přesun dat mezi daty uloženými v souboru a daty v paměti. Nemusíme se tak starat o to, která data a kdy načíst, popř. kdy je uložit na disk.

V unixových operačních systémech pro namapování obsahu souboru do paměti slouží funkce

```
void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);
```

Tato funkce má svůj prototyp v hlavičkovém souboru `sys/mman.h` a její argumenty mají následující význam.

1. Argument `addr` představuje adresu, kam by měla být data namapována (můžeme uvést i hodnotu `NULL`).[1]
2. Argument `len` udává rozsah dat, která mají být namapována do paměti.
3. Argument `prot` udává režim ochrany paměti, přičemž možné hodnoty jsou `PROT_READ` (data je možné číst), `PROT_WRITE` (data mohou být měněna), `PROT_EXEC` (data mohou být vykonána jako program), `PROT_NONE` (k datům není možné přistupovat).[2]
4. Argument `flags` specifikuje, jak se má namapovaná paměť chovat. Jedná se například o příznak `MAP_SHARED`, který zajistí, že změny v datech jsou viditelné i pro jiné procesy. Naopak příznak `MAP_PRIVATE` zajistí, že změny v datech jsou viditelné jen pro proces, který data změnil,[3] což je výhodné zejména v situaci, kdy data pouze čteme a nechceme změny ukládat zpět na disk.

[1] Tato adresa funguje jako nápověda pro jádro OS, protože příslušná oblast může být již využita. Dále pokud adresa není zarovnaná na velikost stránky, data jsou namapována od nejbližší vyšší adresy zarovnané na velikost stránky. Pomocí příznaku `MAP_FIXED` v argumentu `flags` můžeme přimět OS, aby pouze tuto adresu bral jako místo, kam mají být data namapována. V takovém případě ale hrozí, že operace selže.

[2] Soubor, který je namapovaný do paměti musí být otevřený s kompatibilními příznaky.

[3] Používá se technika copy-on-write.

5. Argument `fildes` představuje popisovač souboru, který typicky získáme voláním funkce `open`, viz cvičení č. 5.

6. Argument `off` určuje pozici v souboru, odkud budou data do paměti mapována. Tato pozice musí být násobkem velikosti stránky.[4]

Návratová hodnota funkce je adresa, odkud jsou data k dispozici, případně hodnota `MAP_FAILED`, pokud operace selhala.

### 1.1 Čtení dat ze souboru

Při mapování souboru do paměti, kdy předpokládáme pouze jeho čtení, postupujeme tak, že si nejdříve soubor otevřeme. Pokud jej chceme namapovat do paměti celý, zjistíme jeho velikost a zavoláme funkci `mmap`, jak ukazuje následující kód.

```
// vytvori/otevre soubor
int fd = open("foo.txt", O_RDONLY);
// zjisteni velikosti souboru
struct stat st;
fstat(fd, &st);
size_t length = st.st_size; // velikost dat
// provede namapovani souboru do pameti
char *data = mmap(NULL, length, PROT_READ, MAP_PRIVATE, fd, 0);
if (data == MAP_FAILED) {
   printf("Unable to map file to memory.");
   exit(1);
}
// vypsani obsahu pameti (nemame zarucene, za posledni znak je \0
for (int i = 0; i < length; i++) {
    putchar(data[i]);
}
// provede odmapovani souboru a jeho zavreni
munmap(data, length);
close(fd);
```

Funkce `mmap` vrací adresu, odkud jsou data namapována. Můžeme s nimi pracovat vhodným způsobem. V ukázkovém souboru s nimi pracujeme jako s prostým polem znaků. Práci s mapovanou pamětí ukončíme voláním funkce `munmap`, která se postará o odmapování oblasti paměti v zadaném rozsahu, tj. od předané adresy a v zadané délce.

U funkce `mmap` jsme použili příznaky `PROT_READ` a `MAP_PRIVATE`.

**Úkol č. 1:** Vyzkoušejte, že do dané oblasti paměti nelze zapsat. Kód upravte tak, aby do dané oblasti mohlo být zapisováno. Ověřte, že se změny projevily jen v datech, se kterými pracuje akutální proces, a nejsou promítnuty do vstupního souboru.

[4] Velikost strámky můžeme získat pomocí `sysconf(_SC_PAGESIZE)`.

## 1.2 Zápis a vytvoření souboru

Při zápisu do souboru postupujeme velmi podobným způsobem, jako při mapování souboru pro čtení, jak ilustruje následující příklad.

```
size_t length = 10; // velikost dat
// vytvori/otevre soubor
int fd = open("bar.txt", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
// nastavi souboru zadanou velikost
ftruncate(fd, length);
// provede namapovani souboru do pameti
char *data = mmap(NULL, length, PROT_WRITE, MAP_SHARED, fd, 0);
if (data == MAP_FAILED) {
    printf("Unable to map file to memory.");
    exit(1);
}
// zapise data do dane oblasti pameti
strcpy(data, "123456789");
// provede odmapovani souboru a jeho zavreni
munmap(data, length);
close(fd);
```

Okumentujeme pouze hlavní rozdíly. Pokud chceme soubor i vytvořit, je nutné při jeho otvírání použít příznak `O_CREAT` a uvést oprávnění pro přístup k souboru. V našem případě jsme určili, že pouze vlastník (aktuální uživatel) může do vytvořeného souboru zapisovat a číst jej. Pomocí funkce `ftruncate` určíme velikost souboru.[5]

Při mapování souboru do paměti používáme příznak `PROT_WRITE`, aby bylo možné do souboru zapisovat a `MAP_SHARED`, aby se změny promítly do souboru, se kterým pracujeme. Poznamenejme, že změny se v souboru nemusí objevit okamžitě, mohou se promítnout až v momentě, kdy soubor z paměti odmapujeme. Pokud potřebujeme změny promítnout ihned, můžeme použít funkci `msync`.

Všimněme si, že pro zápis dat do souboru používáme funkci `strcpy`, analogicky bychom mohli s daty pracovat pomocí dalších funkcí včetně `memcpy`, `strcat` nebo `snprintf`.

**Úkol č. 2:** Uvažujme vstupní soubor, který se bude skládat z řádků, kde na každém řádku bude posloupnost čísel v desítkové soustavě, které jsou odděleny mezerami. Napište program, který tento vstupní soubor převede do formátu, který bude obsahovat znaky `'.'` a `'X'`, kde znak `'X'` bude uveden na pozici, která odpovídá číselné hodnotě ze vstupu.

Příklad vstupu:

```
0
0 1
0 2
0 3
```

[5] Tato funkce může velikost souboru zmenšit ale i zvětšit.

```
0 1 2 3 4
```

Příklad výstupu:

```
X....
XX...
X.X..
X..X.
XXXXX
```

Program napište tak, aby pro čtení i zápis dat používal mapování souborů do paměti.

**Bonusový úkol:** Napište druhou variantu programu, která bude používat běžné funkce pro práci se soubory.

## 2 Sdílená paměť

Mapovaní souborů do paměti jde využít i k vytvoření sdílené paměti, která umožňuje komunikovat mezi procesy navzájem. My si takový způsob komunikace představíme na aplikaci, která umožní pomocí sdílené schránky zasílat zprávy mezi procesy. Pro jednoduchost budeme předpokládat, že mezi sebou komunikují pouze dva procesy a v jeden okamžik je možné mít ve schránce maximálně jednu zprávu.

### 2.1 Inicializace

Začneme tím, že deklarujeme strukturovaný datový typ reprezentující schránku.

```
struct mbox {
    sem_t lock;           // sdileny semafor
    char status;          // stav schranky (viz makra DATA_*)
    char data[MBOX_SIZE]; // samotna data
};
```

Schránka má tři atributy. Jednak je to semafor, který řídí přístup ke schránce, dále je to atribut signalizující stav schránky, který může nabývat hodnot – *schránka je prázdna, schránka obsahuje zprávu, komunikace ukončena*. Poslední atribut představuje data uložená ve schránce.

Pomocí mapování souboru do paměti získáme oblast paměti, kde bude tato struktura uložena.

```
size_t length = sizeof(struct mbox);
int fd = open("shm.dat", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
ftruncate(fd, length);
struct mbox *mb = mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

V tomto případě budou data uložena v souboru `shm.dat` a současně se budou nacházet na adrese dané ukazatelem `mb`, který představuje ukazatel na schránku pro zasílání zpráv.

Abychom mohli zajistit korektní komunikaci, je nutné přístup ke schránce synchronizovat a inicializovat její stav. K synchronizaci použijeme semafor, přičemž při inicializaci semaforu uvedeme, že je uložený ve sdílené paměti a slouží k synchronizaci procesů. Viz následující kousek kódu.

```
sem_init(&(mbox->lock), 1, 1);
mbox->status = DATA_UNAVAILABLE;
```

**Poznámka:** Pro jednoduchost předpokládáme, že o inicializaci sdílené paměti a odpovídající datové struktury se stará jeden z procesů, v našem případě je to proces, který do schránky zapisuje.

## 2.2 Zasílání zpráv

Pro jednoduchost budeme předpokládat, že je možné zasílat zprávy jen v podobě textu, který obsahuje číselnou informaci.[6] Proces zasílání zpráv ukazuje následující funkce:

```
1  void mbox_send(struct mbox *mbox, int value){
2      int send = 0;
3      while (!send) {
4          sem_wait(&mbox->lock);
5          if (mbox->status == DATA_UNAVAILABLE) {
6              sprintf(mbox->data, "msg: %i", value);
7              mbox->status = DATA_AVAILABLE;
8              printf("SENDER: %s\n", mbox->data);
9              send = 1;
10         }
11         sem_post(&mbox->lock);
12     }
13 }
```

Tato funkce při každém přístupu ke sdílené schránce schránku uzamče (viz řádky 4 a 11). Pokud schránka neobsahuje data (řádek 5), zapíšeme do schránky zprávu (řádek 6)[7] a nastavíme signalizujeme, že ve schránce jsou data (řádek 7). Pokud není možné zprávu do schránky zapsat, čekáme ve smyčce.

**Poznámka/úkol:** Pokud se chceme podívat na obsah schránky, můžeme použít například příkaz `hexdump -C shm.dat`. Při zkoumání toho, jak vypadá obsah schránky, je vhodné vložit zpoždění mezi odesílané zprávy, např. pomocí `sleep(1)`.

## 2.3 Čtení zpráv

Při čtení zpráv postupujeme analogicky, viz následující kód. Ve smyčce (řádky 4 až 16) čekáme, dokud nebude ve schránce zpráva nebo příznak, že je komunikace ukočena. Každý přístup do schránky je chráněn

[6] Toto řešení bylo zvoleno, aby šlo snadno poznat, že zprávy byly zaslány a převzaty korektně. Úprava pro jiný typ zpráv je přímočará.

[7] Všimněme si použití funkce `sprintf`, která se chová jako `printf`, ale zapisuje výstup do zadaného bufferu.

zámkem (řádky 5 a 15). Pokud je ve schránce zpráva, je její obsah vypsán na standardní výstup,[8] pokud je ve schránce signalizován konec komunikace, je tato informace předána formou návratové hodnoty.[9]

```
1  int mbox_receive(struct mbox *mbox) {
2      int read = 0; // signalizuje uspesne precteni dat
3      int prev_status = 0;
4      while (!read) {
5          sem_wait(&mbox->lock);
6          prev_status = mbox->status;
7          switch (mbox->status) {
8              case DATA_AVAILABLE:
9                  printf("%s\n", mbox->data);
10             case DATA_COMPLETE:
11                 mbox->status = DATA_UNAVAILABLE;
12                 read = 1;
13                 break;
14         }
15         sem_post(&mbox->lock);
16     }
17     return prev_status;
18 }
```

**Poznámka/úkol:** Vyzkoušejte aplikaci. Nejdříve spusťte proces, který bude data do schránky zapisovat a až po něm proces, který bude data číst. Všimněte si, že sdílená paměť je v obou procesech na jiných adresách. Co z toho plyne pro data uložená ve sdílené paměti? Vyzkoušejte, že synchronizace je nutná.

## 2.4 Alternativy pro vytvoření sdílené paměti

Výhodou a současně nevýhodou vytvoření sdílené paměti pomocí namapovaní souboru do paměti je to, že se jednotlivé procesy musí dohodnout na cestě k souboru se sdílenými daty a případně mít možnost zápisu do souborového systému.

### 2.4.1 Objekty sdílené paměti

Alternativou mohou být pojmenované objekty sdílené paměti. Tento objekt získáme pomocí funkce `shm_open`, která má argumenty podobné jako `open` s tím rozdílem, že první argument je jméno objektu, které slouží jako identifikátor sdílené paměti. Použití je velmi podobné funkci `open`.

```
int fd = shm_open("my_shm_file", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
```

[8] V realné aplikaci bychom měli otestovat, jestli jsou ve schránce skutečně očekávaná data, v našem případě řetězec o délce maximálně `MBOX_SIZE` ukončený nulou. Jinak hrozí, že v aplikaci bude bezpečnostní problém nebo bude pro neplatný vstup padat.

[9] Protože převzetí zprávy i ukončení komunikace sdílí stejný kód, není u první větve `switch` záměrně použito `break` a je využito „propadnutí" do další větve.

Tato funkce vrací popisovač souboru, který můžeme namapovat do paměti pomocí `mmap`. Pokud práci s takto sdílenou pamětí ukončíme, je nutné zavolat funkci `shm_unlink`.

**Poznámka:** V Linuxu jsou tyto funkce implementovány tak, že vytvoří soubor v adresáři `/dev/shm/`. Upravte ukázkový kód a ověřte toto tvrzení.

### 2.4.2 Anonymní paměť

Soudobé unixové operační systémy umožňují vytvořit oblast mapované paměti, která není spojena s žádným souborem. Slouží k tomu příznam `MAP_ANON` nebo `MAP_ANONYMOUS`. Takto namapovanou paměť můžeme sdílet mezi rodičem a potomkem nebo potomky navzájem. Je tak možné obejít vlastnost *copy-on-write*, na které je postaveno systémové volání `fork()`.

**Úkol č. 3** Vytvořte třetí typ procesu, který bude kontinuálně sledovat obsah sdílené schránky, a bude zobrazovat informaci, v jakém stavu se schránka nachází (případně obsah zprávy). Tento typ procesu by neměl obsah schránky měnit.

# 3 Ochrana paměti

Mapování paměti se využívá i v případech, kdy chceme mít určité oblasti paměti chráněné proti některým operacím, např. zápis nebo provádění dat jako kódu. Ochranu proti možnosti zápisu jsme viděli v úvodní kapitole, kdy ochrana jednotlivých stránek byla předána jako argument funkci `mmap`, alternativně ke změně ochrany stránek můžeme použít funkci `mprotect`.

## 3.1 Kód jako data a jeho provedení

Uvažujme jednoduchou funkci v assembleru.

```
0: 89 f8                    mov    eax,edi
2: ff c0                    inc    eax
4: c3                       ret
```

Pokud tuto funkci přepíšeme do strojového kódu a zavoláme, měla by se nám vrátit hodnota o 1 větší. K zavolání tohoto strojového kódu bychom mohli použít následující kód v C.

```
1 typedef int (*intfun)(int);
2 char data[] = { 0x89, 0xf8, 0xff, 0xc0, 0xc3};
3 int main() {
4     intfun f = (intfun) data; // FAIL
5     printf("%i\n", f(10));
6 }
```

Pro přehlednost na řádku č. 1 definujeme typ ukazatel na funkci, která má právě jeden argument typu `int` a vrací hodnotu typu `int`. Na řádku č. 2 je uložena funkce do pole `data` v podobě strojového kódu. Na řádku č. 5 toto pole přetypujeme na ukazatel na funkci a funkci zavoláme.

Pokud tento program spustíme, s velkou pravděpodobností dojde k chybě, protože pole `data` je uloženo v oblasti (stránce), kde je zakazané provádění kódu.

Abychom kód mohli vykonat, je nutné jej přesunout do stránek, u nichž je povolené provádění kódu. Ty můžeme získat pomocí mapování souboru do paměti, například následovně.

```
unsigned char *ex_data = mmap(NULL, sizeof(data), PROT_WRITE | PROT_EXEC,
                              MAP_PRIVATE | MAP_ANON, -1, 0);
memcpy(ex_data, data, sizeof(data));
intfun f = (intfun) ex_data;
printf("%i\n", f(10));
```