Operační systémy 2: Praktická cvičení

Petr Krajča

2024

Prolog

Tento text představuje soubor praktických cvičení určených k pochopení a prohloubení znalostí o soudobých operačních systémech. Text je zaměřen na praktické použití rozhraní, která nabízí nejrozšířenější operační systémy, tj. Linux a Windows NT. To zahrnuje problematiku synchronizace procesů a vláken, práci se vstupy a výstupy, mapování souborů do paměti. Závěrečné kapitoly jsou věnovány pokročilým tématům, jako je implementace souborového systému nebo implementace alokátoru paměti.

Obsah

1

Problém synchronizace při programování vícevláknových aplikací

Programování vícevláknových (nebo obecněji paralelních) aplikací přináší celou řadu úskalí, která se při běžném programování nevyskytují. Velmi obvyklé je, že chyby se mohou objevovat nedeterministicky nebo se mohou projevit za velmi specifických okolností a jejich odhalení proto bývá komplikované.

1.1 V čem je problém a jak se mu vyhnout

Typický problém, se kterým se při programování vícevláknových aplikací můžeme setkat, jsou tzv. chyby souběhu (*race condition*), kdy více vláken pracuje souběžně se sdílenými prostředky. Těmi jsou typicky proměnné (příp. objekty v paměti), otevřené soubory, síťová a databázová spojení apod. Problém nastává, pokud jedno nebo více vláken může měnit stav sdílených prostředků,[1] např. když mění hodnotu globální proměnné nebo sdíleného objektu, zápisuje do souboru apod.

Nejefektivnějším způsobem, jak předejít chybám souběhu, je vyhnout se jim zcela. Víme-li, že potenciálním zdrojem chyb jsou změny sdílených prostředků, můžeme:

(a) *Vyhnout se sdílení*, tj. pokud je to z povahy problému možné, vytvoříme kopii dat, se kterými pracuje jen dané vlákno. To sice přináší režii, ale vyhneme se režii, kterou by měla koordinace přístupu k těmto datům.[2]

(b) *Vyhnout se změně stavu*, tj. máme-li data (nebo prostředky), která jsou od určitého bodu sdílena mezi dvěma a více vlákny, neměli bychom je měnit.

Jinými slovy je dobré, pokud můžeme vláknu předat na vstupu všechna data, která bude pro svou činnost potřebovat, tak aby vlákno nemuselo nijak interagovat s ostatními vlákny. U takto navrženého programu

[1] V angličtině se to označuje jako *shared mutable state*.

[2] Toto řešení může na první pohled vypadat nepřirozeně, protože vyžaduje paměť navíc a programátoři mají (nebo by měli mít) tendenci zbytečně neplýtvat prostředky.

nejenže předejdeme některým typům chyb, ale program bude snazší na pochopení, což je žádaná vlastnost.

Jsou však situace, kdy se změnám sdíleného stavu nemůžeme vyhnout. V takovém případě je nutná *vzájemná synchronizace* vláken při přístupu ke sdíleným prostředkům. Tato synchronizace musí být zajištěna kdykoliv nějaké vlákno *mění sdílené prostředky* nebo *je čte*. Protože práce se sdílenými prostředky je místem, kde dochází nejčastěji k chybám souběhu, je vhodné, práci se sdílenými prostředky soustředit do malého množství funkcí (nebo metod), u kterých můžeme snadno ověřit, že máme korektně synchronizovaný přístup ke sdíleným prostředkům, a tak garantovat správný běh programu.

1.2 Synchronizace

1.2.1 Testovací prostředí

Problematiku synchronizace si ukážeme na tzv. problému producent-konzument. Kdy máme jedno nebo více vláken, která data vytváří (producent), a současně máme jedno nebo více vláken, která data zpracovávají (konzument).[3] Data mezi producenty a konzumenty se předávají přes sdílenou paměť, k čemuž se nejčastěji používá datová struktura fronta (FIFO). Pokud je fronta zaplněna, producent musí počkat, dokud se ve frontě neuvolní místo, než může předat data. Analogicky, pokud je fronta prázdná, čeká konzument na vložení dat do fronty.

V našem demonstračním příkladu použijeme klasickou implementaci fronty, do které je možné vkládat ukazatele, případně čísla přetypovaná na ukazatele, viz soubory `queue.c` a `queue.h`. Dále vytvoříme dvě vlákna producentů, která budou ukládat do fronty posloupnost celých čísel od zadané hodnoty. Kód tohoto vlákna vypadá následovně.

```
1  lock_t queue_lock = LOCK_INITIAL_VALUE;
2
3  void *producer(void *arg)
4  {
5          struct producer_request *req = arg;
6          struct queue *queue = req->queue;
7          int start = req->start;
8          for (int i = start; i < start + TEST_SIZE; i++) {
9                  int result;
10                 do {
11                         lock(&queue_lock);
12                         result = queue_put(queue, (void *) i);
13                         unlock(&queue_lock);
14                 } while (result == Q_FULL);
15         }
16         return NULL;
17 }
```

[3] Tato architektura je praktická, pokud potřebujeme zpracovat větší množství dat a rozložit zátěž mezi více procesorových jader.

Vlákno převezme odkaz na objekt s frontou (řádek 6) a hodnotu, od které má ukládat čísla do fronty (řádek 7). Samotné ukládání probíhá v cyklu na řádcích 8 až 15. Vložení hodnoty je na řádku 12. Všimněme si, že tato operace je jednak obalena cyklem `while`, který zajišťuje opakované vložení, pokud je fronta plná. Dále si povšimněme funkcí `lock` a `unlock`. Funkce `lock` by měla zajistit, že v daný okamžik s frontou nebude pracovat žádné jiné vlákno. Jinými slovy, pokud s frontou již nějaké vlákno pracuje, musí vlákno volající `lock` počkat, dokud jiné vlákno nezavolá `unlock`. A opačně, pokud operace `lock` proběhne, jiná vlákna musí počkat, dokud nedojde k zavolání `unlock`.

Dále vytvoříme jedno vlákno konzumenta, které přečte všechny hodnoty z fronty a vypíše je. Kód tohoto vlákna vypadá následovně.

```
1  void *consumer(void *arg)
2  {
3          struct queue *queue = arg;
4          for (int i = 0; i < 2 * TEST_SIZE; i++) {
5                  int result;
6                  do {
7                          void *value;
8                          lock(&queue_lock);
9                          result = queue_get(queue, &value);
10                         unlock(&queue_lock);
11                         if (result == Q_OK) {
12                                 printf("%i\n", (int) value);
13                                 fflush(stdout);
14                         }
15                 } while (result == Q_EMPTY);
16         }
17         return NULL;
18 }
```

Vlákno konzumenta odebírá hodnoty z fronty (řádek 9), a pokud je odebrána nějaká hodnota (řádek 11 až 14), je vypsána na standardní výstup. Protože předpokládáme, že se při testování budou objevovat chyby, po každém vypsání hodnoty pro jistotu vyprázdníme buffer (řádek 13), abychom měli vypsáno opravdu vše a nezůstalo nám něco v bufferu.

1.2.2 Bez synchronizace

Nejdříve vyzkoušíme, co se stane, pokud synchronizace nebude aktivní, tj. obě funkce pro synchronizaci budou vypadat následovně:[4]

```
void lock(lock_t *lock)
{
```

[4] V přiložených zdrojových kódech jsou uvedeny různé varianty řešení zamykání. Pomocí podmíněného překladu je možné vybrat některou z implementací. Variantu, kdy je zamykání deaktivováno, je možné zvolit pomocí `#define USE_NO_LOCKS`.

```
}
void unlock(lock_t *lock)
{
}
```

Protože s hodnotou zámku nijak nepracujeme, datový typ `lock_t`, můžeme zvolit libovolně, např. použít typ `int`.

```
typedef int lock_t;
```

Úkol 1: Program přeložte a spusťte.

Je pravděpodobné, že program neskončí.[5] Pokud by program běžel správně, měl by vypsat 2× `TEST_SIZE` hodnot. Počet vypsaných hodnot můžeme zjistit připojením nástroje `nl`[6] na výstup testovacího programu, tj. spustíme `./queue-test | nl`.

Vidíme, že počet řádků je menší než 2× `TEST_SIZE`. Při opakovaném spuštění bychom měli vidět, že počet vypsaných řádků se mezi spuštěními liší.

Úkol 2: Promyslete a zdůvodněte, proč program neskončil.

1.2.3 Špatně vyřešená synchronizace

Jako jednoduché řešení problému se nabízí vytvořit si zámek, např. proměnnou typu `int`, kde hodnota 0 bude signalizovat, že zámek je odemčený, a hodnota `1` bude signalizovat, že zámek je zamčený. Pokud je zámek zamčený, mělo by vlákno volající funkci `lock` čekat.

Takto pojaté zamykání by mohlo být implementované následovně.[7]

```
typedef int lock_t;
#define LOCK_INITIAL_VALUE (0)

void lock(lock_t *lock)
{
        while (*lock) { }
        *lock = 1;
}
void unlock(lock_t *lock)
{
        *lock = 0;
}
```

Pokud program vyzkoušíme, měl by se chovat podobně špatně jako v předchozím případě, tj. vypíše na výstup čísla, ale ne všechna.

[5] A je nutné jej ukončit, např. pomocí `Ctl+C`.

[6] Tento nástroj čte řádky ze svého vstupu a vypisuje je očíslované.

[7] Tento typ zamykání v ukázkovém kódu aktivujete pomocí `#define USE_NAIVE_SPINLOCK`

Úkol 3: S pomocí nástroje `objdump`[8] se podívejte, jak byl kód funkce `lock` přeložen, a projděte si jednotlivé kroky (instrukce).

Úkol 4: V souboru `Makefile` změnte úroveň optimalizací, které překladač provádí, tj. zaměnte hodnotu `-O0` za `-O2` v proměnné `CFLAGS`. Odstraňte předchozí přeložené soubory pomocí `make clean` a program znovu přeložte a spusťte.

V této situaci se může stát,[9] že bude program rozbitý novým způsobem. Může se stát, že program nevypíše žádnou hodnotu. Podíváme-li se na to, jak byla funkce přeložena, můžeme v ní vidět přibližně následující.

```
100:   8b 07                    mov    eax,DWORD PTR [rdi]
102:   66 0f 1f 44 00 00        nop    WORD PTR [rax+rax*1+0x0]
108:   85 c0                    test   eax,eax
10a:   75 fc                    jne    108 <lock+0x8>
10c:   c7 07 01 00 00 00        mov    DWORD PTR [rdi],0x1
112:   c3                       ret
```

Na adrese 100 přečteme hodnotu zámku, který je předán pomocí prvního argumentu (viz registr `rdi`), instrukci na adrese 102 můžeme ignorovat, protože instrukce `nop`[10] nic neprovádí. Instrukce na adrese 108 otestuje, zde je hodnota zámku 0, nebo ne.[11] Pokud hodnota není nulová, pokračuje se instrukcí na adrese 108. V opačném případě je změněna hodnota zámku a je proveden návrat z funkce `lock`.

Tento výsledek je dán tím, že překladač předpokládal, že hodnota argumentu `lock` nebude změněna a vytvořil nekonečnou smyčku. Z kódu to tak vyplývá a překladač nemá informaci o tom, že kód dané funkce poběží ve více vláknech současně. Práce s vlákny je řešena na úrovni OS nikoliv překladače.

Řešením je označit proměnnou, která představuje zámek, jako `volatile`. Tím, můžeme signalizovat překladači, že daná proměnná může být měněna z více vláken a neměl by s ní provádět některé optimalizace.

Úkol č. 5: Změňte datový typ zámku z `int` na `volatile int`, program přeložte, spusťte a podívejte se, jak byla funkce `lock` přeložena. Pokud vše proběhlo správně, přehnaná optimalizace by měla být eliminována a program by měl být rozbitý původním způsobem, tj. vypisovat hodnoty, ale ne všechny.

1.2.4 Korektně vyřešená synchronizace

Důvod, proč předchozí řešení nefungovalo správně, je ten, že mezi otestováním, jestli daný zámek je `0`, a jeho nastavením na `1` byla prodleva, kdy mohlo dojít k přepnutí mezi vlákny, a tedy dvě vlákna současně mohla uzamknout jeden zámek a pracovat tak s frontou současně.

Řešením tohoto problému je použití atomických operací, tj. operací, které jsou (z pohledu vnějšího pozorovatele) vždy provedeny v jednom nedělitelném kroce. Jednou z takových operací je atomické prohození hodnoty v paměti s obsahem registru. Na procesorech x86/AMD64 se k tomu používá instrukce `xchg`.[12]

[8] Viz cvičení v letním semestru č. 2.

[9] Ale není to pravidlo.

[10] No operation.

[11] Instrukce `test` je podobná instrukci `and` s tím rozdílem, že výsledek operace se nikam neukládá a jen jsou nasteveny příznaky v příznakovém registru. Jedná se o podobný vztah jako u operací `cmp` a `sub`.

[12] K dispozici jsou i další atomické instrukce, např. `cmpxchg`.

Pokud chceme takovou instrukci v programu použít, musíme buď přejít do assembleru, nebo použít některé z nestandardních rozšíření,[13] které obvykle překladače nabízí. My použijeme druhou variantu, jak ukazuje následující kód.

```
1  typedef volatile int lock_t;
2  #define LOCK_INITIAL_VALUE (0)
3
4  void lock(lock_t *lock)
5  {
6          int key = 1;
7          while (__atomic_exchange_n(lock, &key, __ATOMIC_SEQ_CST)) { }
8  }
9  void unlock(lock_t *lock)
10 {
11         __atomic_store_n(lock, 0, __ATOMIC_SEQ_CST);
12 }
```

V cyklu na řádku 7 funkce `__atomic_exchange_n` atomicky prohodí hodnotu `key` a `lock` a tato funkce vrátí původní hodnotu proměnné `lock`.[14] Z toho plyne, že pokud hodnota `lock` byla `1`, je do této proměnné opět nastaveno `1`, je vrácena hodnota `1` a pomocí prázdného cyklu opakovaně testujeme stav proměnné `lock`. Pokud je hodnota `lock` rovna `0`, je do ní atomicky uložena hodnota `key`, tj. hodnota `1`, funkce vrátí `0` a dojde k ukončení smyčky.

Pro odemčení zámku (řádek 11) použijeme opět atomickou operaci, protože u operace přiřazení nemusí být obecně garantováno její atomické provedení.

Úkol č. 6: Program přeložte,[15] vyzkoušejte a podívejte se, jak byly přeloženy funkce `lock` a `unlock`.

V tento okamžik bychom měli mít korektní řešení, které ale není úplně efektivní, protože všechna čekající vlákna spotřebovávají procesorový čas.

[13] `https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html`

[14] Třetí argument `__ATOMIC_SEQ_CST` udává, jak silné garance synchronizace vyžadujeme, v našem případě vyžadujeme úplné uspořádání operací.

[15] S použitou volbou `#define USE_TRUE_SPINLOCK`.

2

Synchronizační nástroje Linuxu určené pro vícevláknové aplikace

V přechozím cvičení jsme si ukázali, jak lze s pomocí atomických operací implementovat vlastní zámky, které zajistí korektní synchronizaci vláken. V praxi bychom se k takovému řešení měli vyhnout, protože není efektivní a můžeme snadno vytvořit špatně odhalitelnou chybu. Čisté řešení je použít některý z nástrojů, který pro synchronizaci nabízí daný operační systém. V tomto cvičení si představíme nástroje pro synchronizaci vláken, které jsou k dispozici v Linuxu.

2.1 Tradiční zámky

Základní nástrojem, který knihovna `pthreads` nabízí je zámek. Ten je označovaný jako mutex.[1] Práce s mutexy je přímočará. Mutex je reprezentován datovým typem `pthread_mutex_t`. Mutex je možné inicializovat dvěma způsoby, buď pomocí funkce `pthread_mutex_init`:

```
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
```

Použití je následovné:

```
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL); // NULL => vychozi vlastnosti
```

Případně můžeme použít inicializační makro `PTHREAD_MUTEX_INITIALIZER`, které inicializuje mutex s výchozími vlastnostmi, což je ve většině případů žádoucí.[2]

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

S mutexy se pracuje pomocí tří základních operací:

[1] Z anglického *mutual exclusion*, vzájemné vyloučení.

[2] Pomocí atributů zámku lze nastavit, zda zámek může být sdílen mezi procesy nebo jak se mají zámky chovat ve specifických případech, např. pokud je ukončeno vlákno držící daný zámek, pokud je zámek zamčen vícekrát jedním vláknem apod.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Funkce `pthread_mutex_lock` provede uzamčení zámku, případně čeká, než bude zámek odemčen. Funkce `pthread_mutex_trylock` se pokusí uzamčít zámek stejně jako by to udělala předchozí funkce, avšak pokud je zámek držen jiným vláknem, nečeká a vrátí hodnotu `EBUSY`. Funkce `pthread_mutex_unlock` zámek odemče.

Po skončení práce se zámkem bychom se měli postarat o jeho zrušení.

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Úkol 1: Upravte zdrojové kódy z minulého cvičení tak, aby vedle námi vytvořených mechanismů zamykání šlo použít i mutexy z knihovny `pthreads`.

Poznámka: Při práci se synchronizačními nástroji bychom si měli dát pozor na nestandardní situace, např. ukončení vlákna držícího zámek, opakované zamykání apod. V řadě situací může dojít k nedefinovanému chování[3] nebo může dojít k uváznutí (deadlocku). Pokud chceme dělat cokoliv jiného než právě jednou zamčít a právě jednou odemčít zámek, je na místě pročíst podrobně dokumentaci.

2.2 Semafory

O něco univerzálnějším mechanismem pro synchronizaci jsou semafory. Práce s nimi je velice podobná práci se zámky. Máme datový typ `sem_t` reprezentující semafor, inicializační funkci `sem_init`, funkce pracující se semaforem, `sem_wait`, `sem_trywait` a `sem_post` a funkci pro uvolnění semaforu `sem_destroy`. Pozor, deklarace typu a prototypů funkcí se nachází v hlavičkovém souboru `samaphore.h`.

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_destroy(sem_t *sem);
```

Funkce `sem_init` akceptuje jako své argumenty počáteční hodnotu semaforu (argument `value`), případně je možné nastavit příznak, že bude semafor sdílen mezi procesy (argument `pshared`).[4] Pokud máme semafor pro synchronizaci vláken, použijeme hodnotu `0`. Funkce pojmenované `wait` snižují (popř. zkusí snížit) hodnotu semaforu, zamykají semafor, a odpovídají tedy operaci `P`. Funkce `sem_post` zvyšuje hodnotu semaforu, odemyká jej, a odpovídá operaci `V`.

Úkol 2: Rozšiřte úkol z minulého cvičení tak, aby k zamykání šlo použít binární semafor.

[3] Program může spadnout, být v nekonzistentním stavu, případně se to může projevit tak, že v našem testovacím prostředí vše bude fungovat správně, ale u uživatelů se objeví chyba.

[4] Toho je možné docílit s využitím sdílené paměti.

2.3 Bariéry

Vedle zamykání se při programování vícevláknových aplikací často setkáváme s požadavkem na to, aby se vlákna v určitém bodě zastavila do té doby, než ostatní vlákna dokončí svou činnost. Jeden takový scénař by mohl vypadat následovně:

(1) Vlákna odděleně provádí inicializaci výpočtu.

— Čeká se až všechna vlákna dokončí inicializaci.

(2) Každé vlákno provede výpočet se svým daty.

— Čeká se až všechna vlákna dokončí výpočet.

(3) Zpracují se výsledky ze všech vláken.

Pro tento typ úloh nabízí knihovna `pthreads` synchronizační nástroj označený jako *bariéra*. Tento objekt slouží k tomu, aby se vlákno programu zastavilo v určitém místě programu do doby, než daného místa v programu dosáhnou ostatní vlákna. Objekt bariéry je reprezentován typem `pthread_barrier_t` a k jeho inicializaci slouží funkce:

```
int pthread_barrier_init(pthread_barrier_t *barrier, pthread_barrierattr_t *attr,
                         unsigned count);
```

Tato funkce akceptuje jako své argumenty objekt bariéry, atributy[5] a počet vláken, které se mají na dané bariéře zastavit. K zastavení na bariéře slouží funkce:

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Tato funkce zastaví běh vlákna do doby, než celkem `count` vláken zavolá `pthread_barrier_wait`.

Ke zrušení bariéry slouží funkce:

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Poznámka: Bariéry jsou při některých nastaveních překladače nedostupné. Abychom je měli k dispozici, je potřeba definovat standard, pro který je program vytvořený. Toho dosáhneme tím, že před direktivy `#include` vložíme odpovídající makro. V našem případě je potřeba zadat:

```
#define _POSIX_C_SOURCE 200112L /* nebo vyssi */
```

Úkol č. 3: Rozšiřte úkol z předchozího cvičení tak, aby se vlákna producentů při dosažení poloviny vygenerovaných čísel se synchronizovala. Tj., aby vlákno, které dosáhne poloviny vygenerováných čísel, počkalo, dokud druhé vlákno nevygeneruje taktéž polovinu čísel.

[5] Pokud použijeme `NULL`, použije se výchozí nastavení.

2.4 Podmínková proměnná

Běžně se stává, že potřebujeme uspat vlákno do doby, než bude splněna nějaká podmínka. Pro tyto účely nabízí knihovna `pthreads` objekt označovaný jako *podmínková proměnná*.[6] Což je objekt, který umožní uspat vlákno do doby, než je vyslán signál, že došlo ke změně a potenciálně byla daná podmínka splněna.

Podmínková proměnná hodnota je typu `pthread_cond_t`, která je inicializována a rušena podobně jako ostatní synchronizační nástroje.

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_destroy(pthread_cond_t *cond);
```

Práce s podmínkovou proměnnou je komplikovanější, protože vyžaduje kooperaci s mutexem, který chrání podmínku, na jejíž splnění čekáme. Jinými slovy, mutex chrání proměnnou na jejíž změnu čekáme. Pro práci podmínkovou proměnnou máme tři funkce.

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Funkce `pthread_cond_wait` zastaví vlákno v daném bodě a odemče zámek `mutex`, který by měl být před zavoláním této funkce ve stavu zamčeno. Funkce `pthread_cond_signal`, probudí jedno z vláken, které na dané podmínkové proměnné čekají, a zamkne mutex, který byl předaný funkci `pthread_cond_wait`. Funkce `pthread_cond_broadcast` se chová obdobně, avšak probudí všechna vlákna čekající na dané podmínkové proměnné.

Scénář použití by měl vypadat přibližně následovně.

```
1  // globalni hodnota predstavujici nejakou podminku
2  volatile int podminka = 0;
3
4  // zamek chranici podminku
5  pthread_mutex_t lock = PTHTREAD_MUTEX_INITIALIZER;
6
7  // podminkova promenna
8  pthread_cont_t cond = PTHREAD_COND_INITIALIZER;
9
10 // vlakno A:
11 pthread_mutex_lock(&lock);
12 while (!podminka) {
13    pthread_cond_wait(&cond, &lock);
14 }
15 pthread_mutex_unlock(&lock);
```

[6] Anglicky *condition variable*.

```
16
17 // vlakno B:
18 pthread_mutex_lock(&lock);
19 podminka = 1;
20 pthread_mutex_unlock(&lock);
21 pthread_cond_signal(&cond);
```

Na řádcích 1 až 8 máme inicializaci. Zámek `lock` slouží k ochraně globální proměnné `podminka`.

Vlákno (A) na řádku 11 zamkne tento zámek a na řádku 12 otestujeme, zda je podmínka splněna. Pokud není, funkce `pthread_cond_wait` na řádku 13 uspí dané vlákna a odemče zámek `lock`. To je zásadní z toho důvodu, aby jiné vlákno mohlo změnit hodnotu proměnné `podminka`.

Vlákno (B) na řádcích 18 až 20 změní stav globální proměnné. Zavoláním funkce `pthread_cond_signal` na řádku 21 probudíme jedno vlákno, které čeká na podmínkové proměnné `cond`. V našem případě je to vlákno označené (A).

Vlákno (A) pokračuje řádkem 13, kdy dojde k probuzení vlákna a současně zamčení zámku `lock`, a můžeme otestovat, zda již byla podmínka splněna, či nikoliv. Pokud podmínka nebyla splněna, opět čekáme na řádku 13, jinak se smyčka ukončí a můžeme pokračovat dalším kódem.

Úkol č. 4: Ukázkový příklad se dvěma producenty a jedním konzumentem by šel vylepšit, protože vlákna neustále soupeří o zámek chránící frontu. A to i v případě, že je fronta plná. Řešení lze vylepšit tak, že pokud je fronta plná, vlákna producenta uspíme s pomocí podmínkové proměnné, a když se místo uvolní, probudíme jej zasláním signálu.

Poznámka: Úlohu zkuste vyřešit samostatně, případně můžete nahlédnout do přiloženého kódu.

2.5 Další nástroje

2.5.1 Spinlock

Implementace mutexů v Linuxu kombinuje aktivní a pasivní čekání, tj. na začátku proběhne pokus získat zámek pomocí spinlocku, a pokud vlákno neuspěje, je uspáno, což na jednu stranu šetří procesorový čas (pokud by mělo vlákno delší dobu čekat), ale zároveň má svou režii (přepnutí vláken). Kombinace aktivního a pasivního čekání dáva smysl pro spoustu běžných úloh. Mohou nastat situace, kdy ale tato kombinaci přináší horší výsledky, zejména, trváli zamčení krátkou dobu. V takovém případě se může hodit použít explicitně spinlock. Pro tyto případy knihovna `pthreads` nabízí implementaci spinlocku, která má podobné rozhraní jako mutexy.

Pro reprezentaci tohoto typu zámku máme datový typ `pthread_spinlock_t`, a pracujeme s nimi pomocí operací.

```
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
int pthread_spin_destroy(pthread_spinlock_t *lock);
```

Úkol č. 5: Rozšiřte úkol z minulého cvičení tak, aby k zamykání šlo použít spinlocky z knihovny `pthreads` a srovnejte jejich rychlost s předchozími řešeními.

2.5.2 Zámky pro čtení a zápis

Pro situace, kdy máme data, která mohou být v jeden okamžik čtena více vlákny, ale zápis je realizován v jeden okamžik právě jedním vláknem, má knihovna `pthreads` samostatný typ zámků `pthread_rwlock_t`. Pracuje se s ním podobně jako s mutexy nebo spinlocky, máme inicializační funkci `pthread_rwlock_init`, funkci pro uvolnění zámku `pthread_rwlock_destroy` a funkce pracující se zámkem `pthread_rwlock_rdlock`, `pthread_rwlock_wrlock`, `pthread_rwlock_unlock`. Přirozeně musíme mít dva typy zamykacích funkcí, které se liší tím, jestli chceme zámek použít v režimu pro čtení nebo pro zápis.

3

Synchronizační nástroje Windows určené pro vícevláknové aplikace

Operační systém Microsoft Windows poskytuje ucelenou a konzistentní sadu nástrojů pro synchronizaci procesů a vláken. Zajímavým rysem této sady je, že se zde využívá i principů objektově orientovaného programování, na kterých je jádro Windows NT postaveno.

3.1 Obecné principy

3.1.1 Stavy objektů

Synchronizační objekty ve Windows se nachází ve dvou stavech. Buď je objekt (i) *signalizovaný* (signaled), nebo (ii) *nesignalizovaný* (non-signaled). Pokud se objekt nachází ve stavu *nesignalizovaný*, znamená to, že je nedostupný, a je potřeba čekat, dokud se nepřepne do stavu *signalizovaný*. Máme-li například zámek (mutex), který je odemčený, nachází se ve stavu signalizovaný, pokud je zámek uzamčený, je ve stavu nesignalizovaný. Podobně, máme-li semafor, který má nenulovou hodnotu, je ve stavu signalizovaný, pokud má semafor hodnotu nula, je ve stavu nesignalizovaný.

Takto obecně navržený synchronizační mechanismus má tu výhodu, že můžeme používat stejné funkce pro různé synchronizační objekty,[1] a můžeme používat širší škálu typů objektů a nemusíme se omezovat na základní synchronizační objekty jako je zámek nebo semafor. Například objekty reprezentující vlákno nebo proces jsou taktéž synchronizační objekty, pokud běží, nachází se v nesignalizovaném stavu, v momentě kdy skončí, přechází do signalizovaného stavu. Takto je možné čekat na dokončení vlákna nebo procesu.

Poznámka: Mezi další objekty, které lze použít k synchronizaci patří třeba vstup z konzole. Pokud je vstup prazdný, je objekt nesignalizovaný, jinak je signalizovaný. Podobně se dá sledovat i dostupnost nebo změny dalších zdrojů, jako je dostupná paměť nebo změny v adresářích.

[1] De facto se jedná o polymorfismus.

3.1.2 Univerzální čekací funkce

Rozhraní Windows NT těží z toho, že všechny synchronizační objekty mají přibližně stejné rozhraní, a proto pro synchronizaci používají relativně malé množství funkcí, které zajišťují, že je běžící vlákno pozastaveno do doby, než je splněna předpokládaná podmínka pro synchronizaci.

S jednou z těchto funkcí jsme se již setkali. Připomeňme práci s vlákny, kterou jsme si nastínili v předchozím semestru.

```
HANDLE *hThread = CreateThread(/* parametry */);
/* dalsi kod */
WaitForSingleObject(hThread, INFINITE);
```

Pro nás je zajímavý poslední řádek, kde je použita funkce `WaitForSingleObject`,[2] která čeká dokud objekt (v našem případě vlákno `hThread`) nepřejde z nesignalizovaného stavu do signalizovaného. Druhý argument funkce představuje maximální čas v milisekundách, po který se na daný objekt čeká, konstanta `INFINITE` indikuje, že se čeká nekonečně dlouho.

Další funkcí, která se používá pro čekaní, je funkce `WaitForMultipleObjects`,[3] která pracuje s více synchronizačními objekty současně a podle zvolených parametrů buď čeká, než budou všechny předané objekty v signalizovaném stavu,[4] nebo čeká, dokud alespoň jeden objekt nebude v signalizovaném stavu, viz dokumentace.

Poslední užitečnou synchronizační funkcí, kterou si zmíníme je funkce `SignalObjectAndWait`,[5] tato funkce přepne objekt do signalizovaného stavu (např. odemče zámek) a čeká, dokud se jiný objekt nepřepne taktéž do signalizovaného stavu.

3.2 Tradiční zámky

3.2.1 Mutex

Základním synchronizačním nástrojem je zámek, který se podobně jako v knihovně `pthreads` označuje jako *mutex*. K vytvoření zámku slouží funkce:[6]

```
HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpMutexAttributes, BOOL bInitialOwner, LPCSTR lpName);
```

První argument udává vlastnosti zámku (může být `NULL`), druhý argument udává, zda je zámek při svém vytvoření uzamčen daným vláknem a třetí argument udává jméno zámku (může být `NULL`). Funkce vrací odkaz typu `HANDLE` pro přístup k danému zámku.

Pro zamčení zámku se používá některá z čekacích funkcí, např. `WaitForSingleObject`, k odemčení slouží funkce `ReleaseMutex`[7] a k uvolnění objektu z paměti se používá funkce `CloseHandle`,[8] která uvolní daný

[2] `https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-waitforsingleobject`
[3] `https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-waitformultipleobjects`
[4] Může sloužit jako bariéra.
[5] https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-signalobjectandwait
[6] `https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-createmutexa`
[7] `https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-releasemutex`
[8] `https://learn.microsoft.com/en-us/windows/win32/api/handleapi/nf-handleapi-closehandle`

objekt, pokud neexistuje na daný objekt žádný odkaz v podobě používané hodnoty typu `HANDLE`.

Práci se zámky ilustruje následující kód:

```
HANDLE hMutex = CreateMutex(NULL, FALSE, NULL); // otevreny zamek s vychozim nastavenim
WaitForSingleObject(hMutex, INFINITE); // zamceni zamku
/* kod kriticke sekce */
ReleaseMutex(hMutex); // odemceni zamku
CloseHandle(hMutex); // uvolneni objektu zamku
```

Úkol č. 1: Vezměte zdrojové kódy z předchozích cvičení a převeďte je na platformu Windows. Kompatibilitu s Linuxem není nutné zachovávat, pro synchronizaci použijte objekt(y) typu mutex.

Poznámka: Windows bohužel nemají v základu ekvivalent nástroje `nl`, kterým bychom mohli otestovat, zda jsou opravdu vypsány všechny řádky. Můžeme si ale pomoci PowerShellem a následujícím kouskem kódu.

```
$i = 1; (.\queue-test.exe).ForEach({ "$i $_i"; $i = $i + 1; });
```

Tento kousek kódu si udržuje proměnnou `$i` obsahující počet vypsaných řádků a pro každý vypsaný řádek programu `queue-test.exe` sestaví řetězec ve tvaru `"$i řádek"`.

Poznámka: Jeden z argumentů, který jsme při vytváření zámku mohli předat, je jméno objektu. Podle tohoto jména může jiný proces, pomocí funkce `OpenMutex`[9] získat přístup k tomuto synchronizačnímu objektu, a je tak možné synchronizovat vlákna napříč procesy.

3.2.2 Kritická sekce

Objekt mutex je navržený a implementovaný jako obecný synchronizační nástroj pro vlákna, který je možné sdílet i mezi procesy, a to může mít dopad na výkon. Proto Windows obsahují ještě jeden nástroj, označovaný jako *kritická sekce* (anglicky *critical section*), který umožňuje synchronizovat pouze vlákna v rámci jednoho procesu, ale měl by být efektivnější.

S kritickou sekcí se pracuje mírně jinak než s mutexy. Jednak synchronizační objekty jsou reprezentovány jako hodnoty datového typu `CRITICAL_SECTION`.[10] Objekt je inicializován pomocí funkce `InitializeCriticalSection`,[11] uzamčen funkcí `EnterCriticalSection`,[12] odemčen funkcí `LeaveCriticalSection`[13] a uvolněn funkcí `DeleteCriticalSection`,[14] kde všechny tyto funkce mají jedinný parametr, kterým je odkaz na hodnotu typu `CRITICAL_SECTION`.

Použití je následující:

```
CRITICAL_SECTION lock; // alokace objektu zamku
InitializeCriticalSection(&lock); // inicializace zamku
```

[9] `https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-openmutexw`
[10] Všimněme si, že se nejedná o `HANDLE`.
[11] `https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-initializecriticalsection`
[12] `https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-entercriticalsection`
[13] `https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-leavecriticalsection`
[14] `https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-deletecriticalsection`

```
EnterCriticalSection(&lock); // zamceni zamku
LeaveCriticalSection(&lock); // odemceni zamku
DeleteCriticalSection(&lock); // uvolneni zamku
```

Úkol č. 2: Upravte kód tak, aby místo zámku typu *mutex* používal *kritickou sekci*.

3.3 Semafor

Vedle jednoduchých zámků můžeme ve Windows pro synchronizaci používat i semafory. Práce s nimi je podobná jako s mutexy. Objekt semaforu vytvoříme pomocí funkce `CreateSemaphore`,[15] kde specifikujeme atributy semaforu, počáteční a maximální hodnotu semaforu a případně název. Ke snížení hodnoty semaforu slouží čekací funkce, např. `WaitForSingleObject`, ke zvýšení hodnoty slouží funkce `ReleaseSemaphore`,[16] kde specifikujeme, o kolik se má zvednout hodnota semaforu a je možné získat i předchozí hodnotu, viz dokumentace.

Úkol č. 3: Upravte kód tak, aby místo zámku typu *mutex* používal *semafor*.

3.4 Další synchronizační nástroje

Mezi další nástroje, které operační systém Windows nabízí patří bariéry. Práce s nimi je velmi podobná tomu, co jsme viděli v Linuxu, a rozhraní pro práci s nimi je velmi podobné rozhraní, které má objekt kritické sekce, viz dokumentace.[17] Analogicky fungují i podmínkové proměnné.[18]

Úkol č. 4: Upravte kód tak, aby se vlákna producentů při dosažení poloviny vygenerovaných čísel se synchronizovala. Tj., aby vlákno, které dosáhne poloviny vygenerováných čísel, počkalo, dokud druhé vlákno nevygeneruje taktéž polovinu čísel.

3.4.1 Čekání na obecnou událost

Operační systém Windows nabízí praktický nástroj, který umožňuje čekat na obecnou událost. Slouží k tomu objekt *event*, který se podle svého nastavení nachází v signalizovaném nebo nesignalizovaném stavu.

K vytvoření objektu slouží funkce `CreateEvent`,[19] kde podobně jako v případě práce s mutexem specifikujeme atributy a název objektu a vedle toho specifikujeme, zda se objekt po vytvoření nachází v signalizovaném nebo nesignalizovaném stavu, a zda je potřeba manuálně objekt resetovat do nesignalizovaného stavu, nebo se tak bude dít automaticky.

S objektem události se pracuje pomocí funkcí `SetEvent`,[20] která přepne objekt do signalizovaného stavu, `ResetEvent`,[21] která přepne objekt do nesignalizovaného stavu, a pomocí některé z čekacích funkcí, např.

[15] `https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createsemaphorea`
[16] `https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-releasesemaphore`
[17] `https://learn.microsoft.com/en-us/windows/win32/sync/synchronization-barriers`
[18] `https://learn.microsoft.com/en-us/windows/win32/sync/condition-variables`
[19] `https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-createeventa`
[20] `https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-setevent`
[21] `https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-resetevent`

`WaitForSingleObject`. Pokud jsme při vytváření objektu nastavili, že se má automaticky resetovat do nesignalizovaného stavu, čekací funkce jej přepne do nesignalizovaného stavu v momentě, kdy některé z vláken projde přes funkci `WaitForSingleObject`.

Úkol č. 5: Rozšiřte předchozí kód o další vlákno, které bude s pomocí objektu event čekat, než některé z vláken producentů dosáhne tří čtvrtin zpracovaných hodnot, a vypíše o tom informaci na standardní výstup.

Bonusový úkol: Nahraďte bariéru z úkolu č. 4 objektem (nebo objekty) typu event.

4

Základní meziprocesová komunikace v Linuxu I: Signály

4.1 Signály

Mezi nejzákladnější nástroje pro meziprocesovou komunikaci v unixových operačních systémech patří tzv. signály. Jedná se o sadu zpráv, které mohou být procesu (nebo některému z jeho vláken) asynchronně zaslány a proces na ně může zareagovat. Signály se používají k ukončení procesu, k ošetření chybových stavů nebo základní správě procesů. Do jisté míry se podobají přerušením, jak je používá hardware.

Je nanejvýš pravděpodobné, že každý, kdo pracoval v Linuxu s programy v příkazové řádce, se signály již setkal. Příkladem budiž kombinace kláves `ctrl+c`, která pošle procesu signál, který se označuje `SIGINT` a znamená, že by měl proces ukončit prováděnou činnost. Vychozí chování je, že se program ukončí. Toto chování lze ale změnit a některé programy toho využívají. Například máme-li interpreter programovacího jazyka a pracujeme-li s ním přes rozhraní typu, REPL[1] může kombinace `ctrl+c` resp. zaslání signálu `SIGINT` přerušit prováděný výpočet, aniž by došlo k ukončení samotného interpretru.

Pro explicitní ukončení procesu slouží signál `SIGKILL`, který obstará ukončení procesu na úrovni jádra OS. Avšak tato forma ukočení je často příliš radikální, protože daný proces nemá šanci na své ukončení jakkoliv zareagovat, například nemůže uložit rozpracovaná data. Proto by se tato forma ukončení procesu měla používat až v případě, že všechny ostatní možnosti selžou. Pro korektní ukončení procesů existuje signál `SIGTERM`, po jehož obdržení by proces měl provést nezbytné kroky pro své ukončení a sám se ukončit.

Další běžný signál, se kterým se setkávají zejména programátoři, kteří vytváří programy v jazyce C/C++ je signál `SIGSEGV`,[2] kterým je procesu oznámeno, že provedl neplatný přístup do paměti a měl by na to zaragovat.[3] Dalším signálem, se kterým se můžeme setkat, je např. signál `SIGSTOP`, který způsobí zastavení (uspání) procesu do doby než obdrží signál `SIGCONT`. Signál `SIGSTOP` je možné procesu zaslat pomocí kombinace kláves `ctrl+z`.

[1] Read-Eval-Print-Loop

[2] Segmentation violation

[3] Výchozí chování je vypsání chybové hlášky *Segmentation fault* a ukončení procesu.

Z uživatelského pohledu se k zasílání signálů používají buď klávasové zkratky (viz předchozí odstavce) nebo nástroj (program) `kill`. Program `kill` akceptuje jako své argumenty signál a id procesu (pid), kterému má být signál zaslán. Signál je určen buď svým číslem nebo pojmenováním. Seznam signálů lze získat příkazem `kill -L`.

Zaslání signálů:

```
kill -s <nazev-signalu> <process-id>
kill -n <cislo-signalu> <process-id>
```

Příklady:

```
kill -s SIGTERM 42
kill -n 9 7475
```

Identifikátor procesu můžeme získat např. s pomocí nástroje `ps` nebo `top`. Případně můžeme použít nástroj `killall`, kde se proces identifikuje pomocí jeho názvu a signál je zaslán všem procesům daného jména.

Úkol č. 1: Vytvořte program, který bude v nekonečné smyčce vypisovat na standardní výstup přirozená čísla. Mezi výpisy jednotlivých hodnot vložte pauzu jednu až dvě sekundy pomocí funkce `sleep`. Tento program spusťte a zkuste zaslat tomuto procesu různé signály `SIGTERM`, `SIGQUIT`,[4] `SIGKILL`, `SIGSTOP`, využijte pro tyto účely klávesové zkratky i příkaz `kill`.

Úkol č. 2: Spusťte nástroj `ping`, který bude testovat dostupnost počítače na nějaké IP adrese (např. 127.0.0.1). Zkuste tomuto procesu zaslat signály jako v předchozím úkolu a sledujte rozdíly.

4.2 Definice vlastních reakcí na signály

Většina signálů má definované své implicitní chování (např. ukončení procesu) nebo je ignorována. Avšak s výjimkou některých konkrétních signálů, jako jsou `SIGKILL`, `SIGSTOP` nebo `SIGCONT`,[5] je možné definovat funkce, které jsou zavolány, pokud proces obdrží zadaný signál. Např. můžeme proces bezpečně ukončit po obdržení signálu `SIGTERM` nebo reagovat jiným způsobem, např. nástroj `ping` po obdržení signálu `SIGQUIT` vypíše průběžné statistiky, ale neukončí se.

K určení, co se má provést po obdržení signálu slouží funkce `sigaction`, jejíž prototyp a datavé struktury, se kterými pracuje najdeme v hlavičkovém souboru `<signal.h>`. Funkce, má tři parametry, kde první parametr je číslo signálu, jehož obsluhu chceme definovat, dále následuje odkaz na strukturu, která definuje chování při obdržení signálu, a pomocí třetího parametru můžeme získat předchozí nastavení obsluhy daného signálu.

K definici obsluhy signálu slouží struktura `struct sigaction`, která ma atribut `sa_handler`, což je ukazatel na funkci typu `void foo(int)` a jedná se o funkci, která bude zavolána po obdržení signálu. Jako argument je této funkci předán obdržený signál.

Dále struktura obsahuje atribut `sa_flags`, kde jsou uloženy příznaky upravující chování obslahy signálu. Protože obsluha příchozího signálu je kód programu, který je prováděný jako každý jiný, může proces

[4] Tento signál standardně slouží k ukončení procesu a získání obrazu paměti (core dumpu) a je možné jej vyvolovat pomocí `ctrl+\`.

[5] Tyto signály jsou de facto určené pro operační systém.

obdržet další signál, na který by mělo být zareagováno. V takovém případě ale hrozí chyba souběhu.[6] Aby se tomu dalo zabránit, je možné pomocí atributu `sa_mask` určit, které signály jsou tzv. zamaskovány a jejich zpracování je odloženo do doby, než bude dokončena právě probíhající obsluha signálu.

Použití ukazuje následující příklad:

```
void my_handler(int id)
{
        printf("signal: %i\n", id);
}

// nastaveni obsluhy signalu
struct sigaction sa;
sa.sa_handler = my_handler;
sa.sa_flags = 0;
sigemptyset(&sa.sa_mask);
sigaddset(&sa.sa_mask, SIGINT);

if (sigaction(SIGINT, &sa, NULL) == -1) {
        printf("error: handler not installed\n");
}
```

Úkol č. 3: Do programu z úkolu č. 1 vhodným způsobem začleňte výše zmíněný kód a ověřte jeho funkčnost.

Pokud proces čeká na blokující operaci (např. je uspán funkcí `sleep` nebo čeká na nějakou I/O operaci) a obdrží signál, který není ignorován, je možné, že daná operace bude přerušena, např. čeká se méně než předepsaný počet sekund, operace není dokončena.[7]

Úkol č. 4: Ověřte, že v případě příchozího signálu, funkce `sleep` nedokončí zadanou dobu čekání.

4.3 Další funkce pro práci se signály

- `int kill(pid_t pid, int sig)`

  – zašle signál `sig` procesu s `pid`

- `int raise(int sig)`

  – proces zašle signál `sig` sám sobě

- `int pthread_kill(pthread_t thread, int sig)`

  – vláknu `thread` je zaslán signál `sig`

[6] race condition

[7] V takovém případě je přerušení operace signalizováno návratovou hodnotou. Toto chování je nutné, aby procesy mohly korektně reagovat na příchozí signály. U jednotlivých funkcí OS je vhodné konzultovat s dokumentací, jak se chovají v případě příchozího signálu.

- `int pause(void)`

  – pozastaví běh procesu, dokud neobdrží signál

- `int alarm(unsigned seconds)`

  – za `seconds` zašle aktuálnímu procesu signál `SIGALRM`.

Úkol č. 5: Rozšiřte řešení z úkolu č. 1 (resp. 3) tak, aby po deseti sekundách došlo k záslání signálu `SIGALRM` a vypsání informace o tom, že uplynulo již deset sekund.

Úkol č. 6: Vytvořte program, který jako svého potomka spustí nástroj `ping` (viz úkol č. 2), třikrát po sobě pět sekund vyčká a pošle mu signál `SIGQUIT`, a nakonec mu pošle signál `SIGTERM`.

5

Základní meziprocesová komunikace v Linuxu II: Vstup a výstup programu

5.1 Vstup a výstup programu

5.1.1 Standardní vstup a výstup

V unixových operačních systémech aplikace komunikují s uživatelem pomocí standardního vstupu a výstupu.[1] Někdy se můžeme setkat se zjednodušeným pohledem, že programy vypisují svůj výstup na terminál (např. funkcí `printf`) nebo načítají vstup z terminálu (např. funkcí `scanf`). Takové zjednodušení je ale velmi zavádějící.

Je pravda, že implicitně je standardní vstup a výstup spojen s terminálem, ze kterého byl program spuštěn, ale operační systém (a jeho uživatelské rozhraní) nám umožňují předefinovat, co bude standardní vstupem a výstupem programu. Například následující příkazy ukazují přesměrování standardního výstupu programu `ls` do souboru `foo.dat` a přesměrování standardního vstupu programu `nl` ze souboru `foo.dat`.

```
ls > foo.dat
nl < foo.dat
```

Unixové operační systémy dále umožňují propojit vstupy a výstupy jednotlivých procesů a vytvořit tak „řetěz" procesů, které si mezi sebou předávájí data. Například následují příkaz ukazuje propojení tří příkazů. První z logovacího souboru vybere první sloupec (např. IP adresy), druhý příkaz tyto hodnoty seřadí a třetí spočítá počet unikátních hodnot.

```
cut -d" " -f1 access.log | sort | uniq -c
```

V tomto případě programy `sort` a `uniq` načítají vstupní data ze standardního vstupu a zapisují je (stejně i program `cut`) na standardní výstup. Shell, ve kterém tento příkaz spustíme, zajistí, že programy oddělené

[1] Ve skutečnosti se jedná o dva výstupy – standardní výstup, kam se vypisují zpracovaná data, a standardní chybový výstup, kam se vypisují informace o chybách případně ladící informace.

znakem | (svislá čára) budou po svém spuštění propojeny tak, že jejich standardní vstupy a výstupy budou provázany, aby si předávaly data.

Poznámka: Součástí unixové filozofie je, že máme spoustu malých jednoduchých programů, které dělají jednu věc (řazení hodnot, výběr unikátních prvků) a dělají ji dobře. Komplexnější funkcionalita pak vzniká kompozicí těchto menších programů právě propojením jejich vstupů a výstupů. Všimněme si podobnosti s běžným programováním (zejména funkcionálním), kdy máme jednoduché funkce a jejich skládáním získáváme složitější funkcionalitu. K tradiční unixové filozofii patří i to, že data jsou reprezentována v textové podobě, a proto máme v základu celou řadu nástrojů pro práci s textem, které umožňují filtrovat řádky textu, transformovat je apod.

5.1.2 Rozhraní OS pro práci se soubory

Se soubory v jazyce C obvykle pracujeme pomocí sady funkcí (`fopen`, `fread`, `fprintf`, ...). Tyto funkce jsou součástí standardní knihovny jazyka a lze je používat bez ohledu na to, na jakém operačním systému program poběží. Vedle toho operační systémy poskytují ještě rozhraní, které je specifické pro daný operační systém.[2] Unixové operační systémy nejsou výjimkou a poskytují specifické rozhraní pro práci se soubory,[3] které je podobné tomu ze standardní knihovny jazyka C, ale jsou tam určité rozdíly.

Otevření souboru

K otevření souboru slouží funkce deklarované v hlavičkovém souboru `fcntl.h`:

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Funkce `open` otevře (případně vytvoří) soubor specifikovaný cestou `pathname` a při otevírání souboru se chová podle příznaků v argumentu `flags`. Minimálně bychom měli uvést v jakém režimu soubor otevíráme. To pro běžné datové soubory znamená použít buď příznak `O_RDONLY` (soubor je otevřen pouze pro čtení), `O_WRONLY` (soubor je otevřen pouze pro zápis) nebo `O_RDWR` (čtení i zápis).[4] Dále je možné specifikovat, zda ukazatel pozice v souboru ukazuje na konec souboru (`O_APPEND`), jestli obsah souboru má být odstraněn (`O_TRUNC`) nebo zda má být soubor vytvořen, neexistuje-li (`O_CREAT`).

Otevření souboru pro čtení ilustruje následující příklad:

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
        int fd = open("foo.txt", O_RDONLY);
```

[2] To jsme mohli pozorovat například na cvičení v předchozím věnovaném *Windows API a základní práce s procesy ve Windows*.
[3] Viz cvičení z minulého semestru věnované *Systémovým voláním*.
[4] Tento přehled si nečiní ambice být úplný a podrobnější vysvětlení a popis dalších příznaků lze nalézt v dokumentaci.

```
        if (fd < 0) {
                printf("Unable to open file foo.txt\n");
                exit(1);
        }
        close(fd);
        return 0;
}
```

Všimněme si návratové hodnoty. Funkce `open` vrací celé číslo, to se označuje jako *file descriptor* nebo *popisovač souboru* a identifikuje námi otevřený soubor. Pomocí tohoto čísla budeme se souborem pracovat a odkazovat na něj. V případě, že operace selže, vrací funkce `open` záporné číslo.

Úkol č. 1: Vyzkoušejte program tak, aby otevření souboru jednou selhalo a jednou neselhalo.

K ukončení práce se souborem (a uvolnění popisovače souboru) slouží funkce `close`, která je deklarovaná v hlavičkovém souboru `unistd.h`.

V případě, že by měl být soubor vytvořen, je nutné uvést třetí argument `mode`, který specifikuje oprávnění pro práci s nově vytvořeným souborem.[5] Tato oprávnění jsou definována jako konstanty ve tvaru `S_I[RWX](USR|GRP|OTH)`, kde `R`, `W`, `X` udávají oprávnění pro čtení, zápis, spuštění souboru, a `USR`, `GRP`, `OTH` určuje, jestli se dané opravnění vztahuje k vlastníkovi souboru, skupině nebo všem ostatním. Například příznak `S_IWUSR` značí, že do souboru může zapisovat jeho vlastník, `S_IROTH` značí, že všichni mohou číst soubor apod.

Otevření a vytvoření souboru, pokud neexistuje, pro zápis a čtení s tím, že se předpokládá zápis za konec souboru ukazuje následující volání:

```
int fd = open("foo.txt", O_CREAT | O_RDWR | O_APPEND, S_IRUSR | S_IWUSR);
```

Pokud bude soubor vytvořen, budou nastavena oprávnění tak, že pouze jeho vlastník bude moci do souboru zapisovat a číst z něj, viz oprávnění `S_IRUSR | S_IWUSR`.

Práce se souborem

K práci se souborem máme k dispozici mimo jiné funkce:

```
ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
```

Funkce `write` zapíše do souboru `fd` obsah bufferu `buf`, kde `count` udává počet bytů, které se mají zapsat. Analogicky funkce `read` načte do bufferu `count` bytů ze souboru `fd`. Obě tyto funkce vrací skutečný počet bytů, které se podařilo přečíst nebo zapsat.[6] Funkce `lseek` umožňuje přesunout aktuální ukazatel v souboru na námi zadaný offset.

[5] Pokud tato opravnění neuvedeme, vezmou se libovolná (náhodná) data ze zásobníku.

[6] Návratová hodnota může být menší než `count`, např. pokud v souboru již nejsou žádná data, která by bylo možné přečíst, protože jsme narazili na konec souboru.

Úkol č. 2: Do souboru `foo.txt` zapište libovolný řetezec. Ověřte, že jsou data zapsána na konec souboru.

Alternativně můžeme popisovač souboru pomocí funkce `FILE *fdopen(int fd, const char *mode)` převést na hodnotu typu `FILE *` a pracovat se souborem pomocí funkcí standardní knihovny jazyka C. Pozor, `mode` v tomto případě znamená způsob otevření souboru jako u funkce `fopen` a musí být kompatibilní se způsobem otevření souboru pomocí `open`. K uzavření souboru se používá funkce `fclose`, která obstará i ukončení práce s popisovačem souboru.

Úkol č. 3: Převeďte popisovač souboru `foo.txt` na hodnotu typu `FILE *` a zapište do souboru řetězec pomocí funkce `fprintf`.

5.2 Přesměrování vstupů a výstupů

5.2.1 Práce se soubory

Po spuštění má každý proces k dispozici tři popisovače – *standardní vstup* (0), *standardní výstup* (1), *standardní chybový výstup* (2).

Úkol č. 4: Pomocí funkce `write` zapište vhodný řetězec na standardní výstup.

Tyto popisovače jsou obvykle spojeny s terminálem, ze kterého byl program spuštěn a ze kterého čte vstup nebo na který zapisuje výstup. Jak bylo zmíněno v úvodní kapitole, toto není výhradní způsob práce se standardními vstupy a výstupy. Vstup nebo výstup můžeme přesměrovat z/do souboru, např. jako u příkazu `ls > foo.txt`.

Ke změně popisovače souboru slouží funkce `dup2` deklarována v hlavičkovém souboru `unistd.h`:

```
int dup2(int oldfd, int newfd);
```

Pokud tato funkce proběhne úspěšně, popisovač souboru `newfd` bude ukazovat na soubor, který je dán popisovačem `oldlfd`.

Praktické použití ukazuje následující příklad:

```
1  int fd = open("foo.txt", O_CREAT | O_RDWR | O_APPEND, S_IRUSR | S_IWUSR);
2  if (fd < 0) {
3          printf("Unable to open file foo.txt\n");
4          exit(1);
5  }
6  if (dup2(fd, 1) < 0) {
7          printf("Unable to redirect std. output\n");
8          exit(1);
9  }
10 printf("Hello world\n");
11 close(fd);
```

Na řádku č. 6 funkce `dup2` zajistí, že popisovač souboru s číslem 1 (tj. standardní výstup) bude ukazovat na soubor daný popisovačem `fd`, tj. na soubor `foo.txt`. To znamená, že jakýkoliv další zápis na standardní výstup (viz volání `printf`) bude proveden do souboru `foo.txt`.

Úkol č. 5: Ověřte, že opravdu jakýkoliv další výstup je přesměrován do souboru `foo.txt`. Volání funkce `printf` nahraďte spuštěním příkazu `ls` pomocí volání `exec`.

Úkol č. 6: Upravte řešení předchozího úkolu tak, aby příkaz `ls` spustil potomek vytvořený pomocí systémového volání `fork`. Zajistěte, že přesměrování standardního výstupu je provedeno až v potomkovi a rodič může zapisovat na svůj standardní výstup, tj. výpis se provede na terminál.

5.2.2 Roury

Vytvoření roury

K propojení dvou programů v unixových operačních systémech se používají roury. Rouru vytvoříme pomocí funkce (resp. systémového volání) `pipe`:

```
int pipe(int fildes[2]);
```

Funkce `pipe` jako svůj argument akceptuje pole popisovačů `fildes` o dvou prvcích. Pokud funkce proběhne v pořádku, uloží se do pole `fildes` dva popisovače souborů. Popisovač `fildes[1]` slouží k zápisu do roury (zapisovací konec roury), popisovač `fildes[0]` slouží k čtení dat z roury (čtecí konec roury). Zapíšeme-li do „souboru", který je určen popisovačem `fildes[1]`, jsou tato data uložena do bufferu (v jádře operačního systému). Opačně, při čtení „souboru", který je dán popisovačem `fildes[0]`, jsou data načítána z tohoto bufferu v jádře OS.

Důležité je, že v situaci, kdy vznikne potomek pomocí volání `fork`, mají rodič i potomek přístup ke stejné rouře, kterou spolu mohou komunikavat, jak ukazuje následující příklad.

```
#define BUF_SIZE        1024
int fds[2];
if (pipe(fds) < 0) {
        printf("Unable to create a pipe\n");
        exit(1);
}
pid_t pid = fork();
if (pid < 0) {
        printf("Unable to create child process\n");
        exit(1);
} else if (pid == 0) {
        /* potomek: cte data z roury*/
        close(fds[1]); // uzavren zapisovaci konec roury
        char buf[BUF_SIZE];
        // precteni dat
        ssize_t cnt = read(fds[0], buf, BUF_SIZE);
        close(fds[0]); // uzavren cteci konec roury
        buf[cnt] = '\0';
        if (cnt >= 0) printf("Received:\n%s\n", buf);
} else {
```

```
21          /* rodic zapisuje do roury */
22          close(fds[0]); // uzavren cteci konec roury
23          write(fds[1], "hello\nworld\n", 12); // zapise dat
24          close(fds[1]); // uzavre zapisovaci konec
25  }
```

Na řádcích 2 až 6 je vytvořený objekt roury. Na řádku č. 6 je vytvořen nový proces, kdy řádky 11 až 19 představují kód, který provádí potomek, a řádky 22 až 24 představují kód provedený rodičovským procesem. V tomto ukázkovém příkladu rodičovský proces zapisuje do roury a potomek data z roury načítá, jinými slovy, rodič posílá data potomkovi.

Rodičovský proces nejdříve uzavře nepotřebný (čtecí) konec roury (řádek 22) a do zapisovacího konce roury zapíše data (řádek 23) a zavře i tento konec roury (řádek 24).

Potomek postupuje analogicky. Zavře nepotřebný (zapisovací) konec roury (řádek 13), přečte data z roury (řádek 16), a zavře i čtecí konec roury (řádek 17). Zbylé řádky v kódu potomka slouží k vypsání předaných dat.

Propojení vstupů a výstup rourou

Protože oba konce roury jsou popisovače souboru, můžeme je přesměrovat na standardní vstup programu nebo naopak můžeme přesměrovat standardní výstup do roury.

Přesměrování roury na standardní vstup programu dosáhneme pomocí:

```
dup2(fds[0], 0);
```

Po provedení této funkce bude popisovač standardního vstupu (tj. 0), ukazovat na čtecí konec roury. Jinými slovy, pokud bychom použili funkci (`scanf`), bude číst data přímo z roury.

Nemusíme zůstávat u čtení dat ze souboru. Můžeme pomocí systémového volání `exec` spustit program, jehož standardní vstup bude nastaven na čtecí konec roury. Například následovně.

```
dup2(fds[0], 0);
execlp("nl", "nl", NULL);
```

Úkol č. 7: Upravte ukázkový kód tak, aby potomek přesměroval čtecí konec roury na svůj standardní vstup a následně spustil program `nl`.

Analogicky můžeme postupovat u zapisovacího konce roury a rodičovského procesu. Standardní výstup programu přesměrujeme do roury.

```
dup2(fds[1], 1);
```

V tento okamžik, cokoliv je zapsáno na standardní vstup (např. funkcí `printf`), je zapsáno do roury. Opět můžeme spustit program jehož standardní výstup bude směrován do roury.

```
dup2(fds[1], 1);
execlp("ls", "ls", NULL);
```

Úkol č. 8: Upravte ukázkový kód tak, aby rodič přesměroval svůj standardní výstup do roury a následně spustil program `ls`.

Úkol č. 9: Vytvořte program, kde rodič jako potomka spustí program `ls`. A výstup tohoto programu bude rourou směrován k rodiči, který výstup programu `ls` bude číst a bude na svůj standardní výstup vypisovat název souboru spolu s pořadovým číslem. Např.

```
bar.txt (1)
baz.txt (2)
foo.txt (3)
```

5.2.3 Pojmenované roury

Roury, které jsme si ukázali v předchozí kapitole, jsou omezeny na procesy, které jsou příbuzné, je mezi nimi vztah rodič-potomek nebo se jedná o potomky stejného rodiče. Pokud chceme propojit nepříbuzné procesy, můžeme použít pojmenované roury, což je speciální typ souboru, který se chová jako roura, tj. je možné do něj zapisovat, a následně z něj data číst v pořadí tak, jak do něj byla zapsána. Proto se tento typ souborů označuje jako FIFO.

Pojmenovanou rouru můžeme vytvořit příkazem `mkfifo`, případně funkcí stejného jména.

Následující příkaz vytvoří speciální soubor `foo.fifo`, který se chová jako roura.

```
mkfifo foo.foo
```

Můžeme si to vyzkoušet následovně.

```
echo -e "abc\ncde\nefg" > foo.fifo
```

Tento příkaz zapíše do souboru tři řádky, které můžeme následně přečíst vhodným příkazem, například s pomocí `cat`, `nl`, ...

```
cat foo.fifo
```

Poznámka: Roury jsou určeny k jednosměrné komunikaci právě dvou procesů. Neexistují žádné prostředky, které by znemnožnily jiné použití, např. pokud by někdo chtěl, aby do roury zapisovalo více procesů, není tomu bráněno, avšak chování je v takovém případě nedefinováno. Pravděpodobně dojde k chybě souběhu, a proto by se mělo s rourami pracovat jen očekávaným způsobem.

6

Událostmi řízené programování ve Windows

6.1 První program a příprava projektu

V tomto cvičení si ukážeme, jak vytvořit aplikaci pro OS Windows od úplného začátku. Vytvoříme si proto prázdný projekt pro desktopové aplikace. Ve Visual Studiu pro nový projekt použijeme *Windows Desktop Wizard*, zvolíme *Application type: Desktop Application (.exe)* a *Empty Project*.[1] Aby se předešlo problémům se zobrazením národních znaků, je v případě grafických aplikací vhodné pro reprezentaci textu používat výhradně široké (dvoubytové) znaky, tj. typ `wchar_t` nebo `WCHAR`.[2] To lze vynutit buď nastavením projektu (*Advanced — Use Unicode Character Set*) nebo definicí symbolu `#define UNICODE`, před vložením hlavičkového souboru `windows.h`.

Vstupním bodem grafických desktopových aplikací je funkce `wWinMain`,[3] které je předán handle na aktuální aplikaci (`hInstance`), argumenty (`lpCmdLine`) a příznaky, jak má být zobrazeno hlavní okno aplikace. Aplikaci, která zobrazí jednoduché okno pomocí funkce `MessageBox`,[4] ukazuje následující kód.

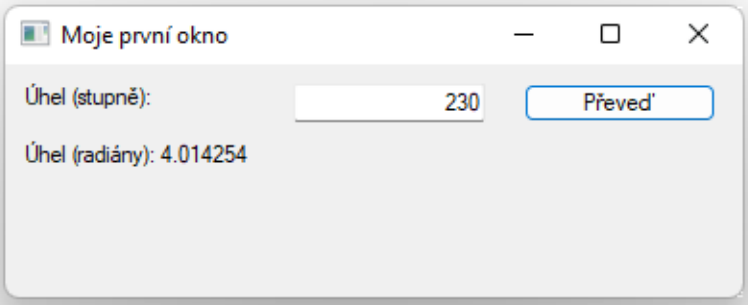
```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR lpCmdLine, int nCmdShow)
{
    MessageBox(NULL, L"Hello world", L"My First Window", MB_ICONEXCLAMATION);
}
```

[1] Tyto volby zajistí, že nebudeme mít projekt zaplněn dalšími předchystanými věcmi, které bychom stejně v tomto cvičení nevyužili.

[2] Použití tohoto typu je vhodné i z toho důvodu, že některé funkce jsou dostupné jen pro tento typ řetězců.

[3] `https://learn.microsoft.com/en-us/windows/win32/learnwin32/winmain--the-application-entry-point`

[4] `https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messagebox`



Obrázek 6.1: Zamýšlený vzhled aplikace

6.2 Složitější aplikace

Nyní si ukážeme, jak vytvořit složitější aplikaci, která se bude skládat z jednoho okna, které bude sloužit k převodu úhlových jednotek, konkrétně k převodu stupňů na radiány. Předpokládaný výsledek můžete vidět na Obrázku 6.1.

6.2.1 Vytvoření okna

K práci s okny, stejně jako k práci s uživatelskými ovládacími prvky, používá operační systém Windows specifický objektový model. Chceme-li vytvořit okno, musíme vytvořit a zaregistrovat jeho třídu, například následovně.

```
1 WCHAR* WND_CLASS_NAME = L"First Wnd Class";
2 WNDCLASS wc = { .hInstance = hInstance,
3                 .lpfnWndProc = WindowProc,
4                 .lpszClassName = WND_CLASS_NAME };
5 RegisterClass(&wc);
```

Na řádku č. 1 definujeme název třídy námi vytvořeného okna, tento název by měl být v rámci aplikace unikátní. Na řádku č. 2 definujeme třídu okna, handle na běžící program, určíme funkci, která bude obsluhovat události spojené s oknem (řádek č. 3), a určíme název třídy (řádek č. 4). Nakonec třídu okna zaregistrujeme (řádek č. 5).

K samotnému vytvoření okna[5] slouží funkce `CreateWindow`,[6] případně `CreateWindowsEx`,[7] která umožňuje nastavit některé dodatečné vlastnosti.

```
1 HWND hwnd = CreateWindow(
2     WND_CLASS_NAME,      // trida okna
```

[5] Vytvoření instance okna v terminologii OOP.
[6] `https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-createwindoww`
[7] `https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-createwindowexw`

```
3       L"Moje první okno",  // nadpis
4       WS_OVERLAPPEDWINDOW, // styl okna
5       CW_USEDEFAULT,       // pozice x
6       CW_USEDEFAULT,       // pozice y
7       400,                 // sirka
8       160,                 // vyska
9       NULL,                // rodicovkseho okno
10      NULL,                // menu
11      hInstance,           // handle aplikace
12      NULL                 // dodatecna data
13  );
```

Všimněme si zejména řádku č. 2, kde k identifikaci třídy okna používáme řetězec, který jsme určili při registraci třídy. Dále si všimněme, že funkce vrací hodnotu typu `HWND`, což je handle, jehož prostřednictvím budeme s oknem pracovat. Jednou z operací, kterou s oknem musíme provést, je jeho zobrazení pomocí funkce `ShowWindow`:

```
ShowWindow(hwnd, nCmdShow);
```

6.2.2 Smyčka událostí

Windows z pohledu grafických aplikací jsou událostmi řízený systém. To znamená, že běžící program získává od operačního systému zprávy o vzniklých událostech, na které pak definovaným způsobem reaguje. Typickým příkladem událostí jsou například stisknutí klávesy nebo kliknutí kurzorem myši na okno programu. Dále pomocí událostí operační systém sděluje, že se okno má překreslit nebo že došlo ke změně rozměrů okna.

Aby program mohl reagovat na jednotlivé příchozí události, obsahuje obvykle funkce `wWinMain` smyčku následujícího formátu:

```
1   MSG msg;
2   while (GetMessage(&msg, NULL, 0, 0) > 0) {
3       TranslateMessage(&msg);
4       DispatchMessage(&msg);
5   }
```

Na řádku č. 2 funkcí `GetMessage`[8] získáme zprávu, na kterou má program zareagovat, např. kliknutí myší nebo požadavek na překreslení okna. Tyto zprávy mohou být dalé transformovány, např. stisk kláves může být převeden na konkrétní událost, viz řádek č. 3 a následně je zpráva předána ke zprácování (řádek č. 4).

[8] `https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getmessage`

6.2.3 Reakce na příchozí zprávy

Funkce `DispatchMessage`[9] předá zprávu příslušnému oknu, které na ni zareaguje funkcí, kterou jsme uvedli při vytváření třídy okna, viz atribut `lpfnWndProc` a hodnota `WindowProc`.[10] Tato funkce je typu:

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
```

První argument udává handle okna, kterého se událost týká, druhý argument představuje identifikátor zprávy, který okno obdrželo, a zbývající dva argumenty představují informace přidružené ke zprávě, např. souřadnice, příznaky, textové informace apod. Význam těchto dvou argumentů je specifický pro každý typ zprávy a je potřeba jej vyčíst z dokumentace.

Funkce `WindowProc` typicky obsahuje jeden `switch`, který se stará o zpracování událostí, na které umí okno zareagovat. Minimálně by to měla být událost `WM_PAINT`, která se stará o překreslení okna. V následujícím ukázkovém příkladě při obdržení zprávy `WM_PAINT` bude obdélníková oblast okna, která má být překreslena, vyplněna barvou okna. Dále při zrušení okna (zpráva `WM_DESTROY`) bude odeslána zpráva `WM_QUIT`, která ukončí smyčku událostí. Pokud nemáme způsob, jak na zprávu zareagovat, použijeme implicitní zpracování pomocí funkce `DefWindowProc`.[11]

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg) {
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    case WM_PAINT:
    {
        PAINTSTRUCT ps;
        HDC hdc = BeginPaint(hwnd, &ps);
        FillRect(hdc, &ps.rcPaint, (HBRUSH)(COLOR_WINDOW));
        EndPaint(hwnd, &ps);
        return 0;
    }
    }
    return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
```

6.2.4 Ovládací prvky

Pokud přeložíme a spustíme dosud popsaný kód, mělo by se nám zobrazit prázdné okno. Do tohoto okna je nutné vložit jednotlivé ovládací prvky. Ovládací prvky, jako jsou tlačítka, textová pole, popisky apod. jsou reprezentovány taktéž jako okna. K jejich vytvoření se proto používá funkce `CreateWindow`. Ukažme si použití na příkladu vytvoření tlačítka:

[9] `https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-dispatchmessage`
[10] https://learn.microsoft.com/en-us/windows/win32/learnwin32/writing-the-window-procedure
[11] `https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-defwindowprocw`

```
1  HWND hwndConvertBtn = CreateWindow(
2      L"BUTTON",   // trida ovladaciho prvku: tlacitko
3      L"Převeď",   // popisek tlačítka
4      WS_TABSTOP | WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,  // styl
5      270,         // pozice x
6      10,          // pozice y
7      100,         // sirka
8      20,          // vyska
9      hwnd,        // rodicovske okno
10     (HMENU)IDBTNCONVERT,  // identifikator vstupu
11     hInstance,
12     NULL);
```

V tomto případě jsme jako třídu okna použili řetězec `"BUTTON"`. Kdybychom chtěli vytvořit textový vstup, použili bychom `"EDIT"`, pro statický popisek `"STATIC"`. Na řádku č. 4 jsou uvedeny vlastnosti tlačítka. Ty zahrnují vlastnosti obecného okna (`WS_*`) a vlastnosti specifické pro konkrétní třídu okna (`BS_*`, `ES_*` apod.)

Umístění jednotlivých ovládacích prvků a jejich rozměry se uvádí jako absolutní hodnoty v pixelech. Na řádcí č. 9 a 11 předáváme odkazy na rodičovské okno a aktuální program.

Důležitý je řádek č. 10, kde uvádíme číselný identifikátor konkrétního ovládacího prvku. Měla by to být číselná hodnota větší nebo rovna 100, která je přetypovaná na typ `HMENU`. Význam této hodnoty bude zřejmý z následujícího textu.

Poznámka: Přiložené zdrojové kódy ukazují vytvoření dalších ovládacích prvků.

6.2.5 Reakce na události spojené s ovládacími prvky

Pokud dojde ke stisku tlačítka, volbě položky z menu, změně textu v textovém vstupu, je rodičovskému oknu zaslána zpráva `WM_COMMAND` a to má možnost na tuto zprávu zareagovat. Abychom mohli určit, od kterého ovládacího prvku zpráva přišla, používá se jako identifikátor hodnota, která je předána funkci `CreateWindow` (viz řádek č. 10 v předchozím výpisu kódu) a tato hodnota se objeví ve `WindowProc` jako parametr `wParam`. Použití v konstrukci `switch` vypadá následovně.

```
case WM_COMMAND:
{
    if (wParam == IDBTNCONVERT) {
        onConvertButtonClick(hwnd);
        return 0;
    }
}
```

6.2.6 Další využití zásílání zpráv

Zasílání zpráv oknům (ať už klasickým oknům nebo ovládacím prvkům) je obecný mechanismus, kterým s okny manipulujeme. Chceme-li například změnit text, slouží k tomu zpráva `WM_SETTEXT`.[12] Tu můžeme oknu explicitně zaslat pomocí funkce `SendMessage`,[13] kdy řetězec předáme pomocí parametru `lParam`, viz dokumentace.

```
SendMessage(hwndResult, WM_SETTEXT, 0, (LPARAM) L"Novy text");
```

Protože manipulace s objekty pomocí explicitního zasílání zpráv není zcela komfortní, existuje řada funkcí, která tento způsob práce s objekty zakrývají. Například pro změnu textu okna existuje funkce `SetWindowText`, která obstará zaslání zprávy v odpovídajícím formátu.

Poznámka: Přiložené zdrojové kódy ukazují, jak lze pomocí zasílání zpráv získat textovou hodnotu. Je to nutné udělat ve třech krocích, nejdříve musíme zjistit velikost textu, pak pro něj alokovat buffer, a nakonec získat text. To se děje zasláním zprávy `WM_GETTEXT`, kde jako `lParam` uvedeme cílový buffer.

6.3 Závěrečné poznámky a úkoly

6.3.1 Zkrocení zpětné kompatibility

Pokud aplikaci spustíme v této podobě, její vzhled bude silně „retro", tj. bude vypadat jako něco z dávné počítačové historie, čti z osmdesátých let. Tento vzhled odpovídá přibližně Windows 3.x nebo 95, viz Obrázek 6.2. Toto chování je důsledkem zachování zpětné kompatibility s předchozími verzemi Windows. Aby aplikace vypadalo soudobě, je nutné načíst novější verzi knihovny s uživatelskými prvky, tj. knihovnu `comctl32` ve verzi minimálně 6.0. K tomu můžeme použít buď nastavení projektu nebo následující direktivy překladače:

```
#pragma comment(lib, "comctl32.lib")
#pragma comment(linker,"\"/manifestdependency:type='win32' \
name='Microsoft.Windows.Common-Controls' version='6.0.0.0' \
processorArchitecture='*' publicKeyToken='6595b64144ccf1df' language='*'\"")
```

Dále je nutné nastavit estetičtější font:[14]

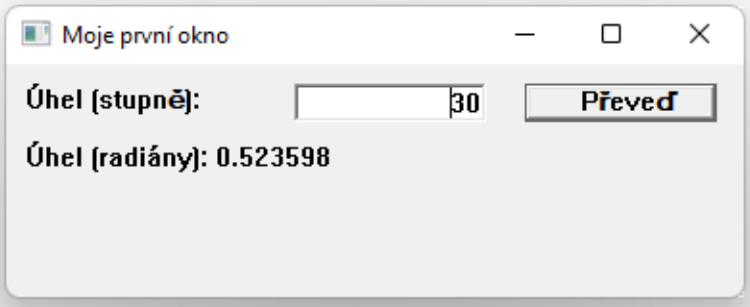
```
HFONT font = CreateFont(12, 0, 0, 0, FW_NORMAL, FALSE, FALSE, FALSE, DEFAULT_CHARSET,
                        OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                        DEFAULT_PITCH | FF_DONTCARE, L"MS Shell Dlg");
SendMessage(hwndInputCaption, WM_SETFONT, (WPARAM)font, TRUE);
SendMessage(hwndInput, WM_SETFONT, (WPARAM)font, TRUE);
SendMessage(hwndConvertBtn, WM_SETFONT, (WPARAM)font, TRUE);
SendMessage(hwndResult, WM_SETFONT, (WPARAM)font, TRUE);
```

Všimněme si, že ke změně fontu je použita zpráva `WM_SETFONT`.

[12] `https://learn.microsoft.com/en-us/windows/win32/winmsg/wm-settext`

[13] `https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-sendmessage`

[14] Ten původní je bitmapový a již v devadesátých letech vypadal hodně archaicky.



Obrázek 6.2: Nemoderní vzhled aplikace

6.3.2 Tvorba skutečných programů

Cílem tohoto cvičení bylo primárně představit mechanismus zasílání zpráv, jak je realizován v operačním systému Windows. Skutečné programy pro Windows se většinou tímto způsobem nevytváří. Definice vzhledu oken, menu apod. se provádí v samostatných souborech, které se označují jako `resources` a tyto soubory jsou vytvářeny typicky pomocí grafických nástrojů jako je Visual Studio, což značně zjednodušuje vývoj.

Poznámka: Vytvořte ve Visual Studiu vzorovou desktopovou aplikaci a podívejte se, jak jsou definovány jednotlivé „resources" a jak se s nimi pracuje z kódu programu.

Protože nativní rozhraní Windows není příliš komfortní, často se nad ním staví samostatné vrstvy, např. v jazyce C++, které programátora odstiňují od spousty technických detailů včetně toho, jak je realizována smyčka událostí a reakce na jednotlivé události. V praxi se dá běžně setkat i s tím, že si aplikace vykreslují obsah okna zcela ve vlastní režii, včetně jednotlivých ovládacích prvků, a z OS si berou jen mechanismus pro zasílání zpráv.

6.3.3 Úkoly

Úkol č. 1: Upravte aplikaci tak, aby se počet radiánů zobrazoval v textovém poli a bylo tam další tlačítko, které umožní převod z radiánů na stupně.

Úkol č. 2: Upravte aplikaci tak, aby testovala, jestli je na vstupu validní hodnota (číslo), a pokud ne, objeví se okno (MessageBox) s chybovou zprávou.

7

Mapování souborů do paměti v Linuxu

7.1 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]

[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.

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.

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.

7.1.1 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.

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

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

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 aktuální proces, a nejsou promítnuty do vstupního souboru.

7.1.2 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);
```

Okomentujeme 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.

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

Příklad vstupu:

```
0
0 1
0 2
0 3
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.

7.2 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.

7.2.1 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.

7.2.2 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)`.

[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.

7.2.3 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 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.

7.2.4 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.

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

[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.

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);
```

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.

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.

7.3 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`.

7.3.1 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));
```

8

Mapování souborů do paměti ve Windows

Při popisu mapovaní souborů do paměti v operačním systému Microsoft Windows se můžeme opřít o znalosti, které jsme získali v případě unixových OS, protože principy zůstavají stejné a jsou zde jen technické, ale přesto zajímavé, rozdíly. Na tyto rozdíly se v tomto cvičení zaměříme především.

8.1 Práce se soubory

Mapování souboru do paměti v případě OS Windows je prováděno ve třech krocích.

1. Nejdříve otevřeme soubor pomocí funkce `CreateFile`.[1]

2. Následně pro tento soubor vytvoříme objekt mapování do paměti pomocí funkce `CreateFileMapping`.[2] Tato funkce má jako své argumenty handle na soubor, otevřený v předchozím kroce, odkaz na bezpečnostní atributy,[3] příznaky ochrany paměti pro namapovanou oblast,[4] maximální velikost namapované oblasti[5] a pojmenování objektu.[6]

3. K samotnému namapování souboru do paměti dojde zavoláním funkce `MapViewOfFile`,[7] které předáme handle na objekt mapování vytvořený v předchozím kroce, příznaky s jakými má být provedeno mapování jednotlivých stránek, začátek oblasti v souboru (offset), která má být namapována do paměti,[8] a velikost oblasti, která má být namapována.

[1] `https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea`

[2] `https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createfilemappinga`

[3] Může být NULL

[4] Např. pouze pro čtení zápis, povolení zápisu, použití copy-on-write. Tyto příznaky musí být kompatibilní s příznaky, které byly použity při otevření souboru.

[5] Tato hodnota je uvedena jako dvě 32bitová slova. Pokud máme existující soubor a uvedeme 0, uvažuje se velikost celého souboru.

[6] To může být NULL, praktické použití uvidíme v dalších příkladech.

[7] `https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-mapviewoffile`

[8] Tento offset je předán jako dvě 32bitová slova a musí být násobkem velikosti stránky.

8.1.1 Čtení dat ze souboru

Následující kód ukazuje mapování obsahu existujícího souboru do paměti.

```
#include <windows.h>
#include <tchar.h>
int _tmain() {
    TCHAR* inputFile = _T("foo.txt");
    DWORD length = 20;
    HANDLE hFile = CreateFile(inputFile, GENERIC_READ, FILE_SHARE_READ, NULL,
                              OPEN_EXISTING, 0, NULL);
    if (hFile == INVALID_HANDLE_VALUE) {
        _tprintf(_T("Unable to open: %s\n"), inputFile);
        return 1;
    }
    HANDLE hMapping = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0, NULL);
    if (hMapping == NULL) {
        _tprintf(_T("Unable to create file mapping\n"));
        _tprintf(_T("last error: %i"), GetLastError());
        return 1;
    }
    LPVOID data = MapViewOfFile(hMapping, FILE_MAP_READ, 0, 0, length);
    if (data == NULL) {
        _tprintf(_T("Mapping failed\n"));
        return 1;
    }
    // prace s daty
    char* text = (char*)data;
    //text[0] = 'X';
    _tprintf(_T("%lx\n"), data);
    for (int i = 0; i < length; i++) {
        _tprintf(_T("%c"), text[i]);
    }
    // konec prace s daty
    UnmapViewOfFile(data);
    CloseHandle(hMapping);
    CloseHandle(hFile);
    return 0;
}
```

Tento kód přesně kopíruje výše popsaný postup a namapuje do paměti obsah souboru pouze pro čtení. Jednotlivé operace mohou z mnoha důvodů selhat. Pozor, funkce `CreateFile` při svém selhání vrací hodnotu `INVALID_HANDLE_VALUE`, funkce `CreateFileMapping` a `MapViewOfFile` vrací `NULL`. Častým důvod selhání jednotlivých funkcí je nekompatibilní nastavení oprávnění/příznaků pro přístup k paměti a sou-

borům. Při hledání příčiny selhání nám může být nápomocna funkce `GetLastError()`, která vrací kód chyby, a seznam jednotlivých chybových kodů.[9]

Pokud bychom chtěli mít možnost upravovat data v paměti, aniž by došlo ke změně souboru, jinými slovy použít techniku copy-on-write, použili bychom na řádku 18 příznak `FILE_MAP_COPY`.

```
LPVOID data = MapViewOfFile(hMapping, FILE_MAP_COPY, 0, 0, length);
```

Po skončení práce s pamětí je nutné oblast odmapovat pomocí funkce `UnmapViewOfFile` a uzavřít všechny otevřené handely.

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 aktuální proces, a nejsou promítnuty do vstupního souboru.

8.1.2 Zápis a vytvoření souboru

Při práci se souborem v režimu pro čtení i zápis pracujeme analogicky jako v případě čtení dat ze souboru. Tj. otevřeme vhodným způsobem soubor:

```
HANDLE hFile = CreateFile(inputFile, GENERIC_READ | GENERIC_WRITE,
                          FILE_SHARE_READ, NULL, CREATE_ALWAYS, 0, NULL);
```

Vytvoříme objekt mapování s právem zápisu:

```
HANDLE hMapping = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, length, NULL);
```

A provedeme namapování do paměti, opět s příznakem povolujícím zápis do dané oblasti paměti.

```
LPVOID data = MapViewOfFile(hMapping, FILE_MAP_WRITE, 0, 0, length);
```

Následně můžeme s pamětí pracovat pomocí libovolných vhodných operací, např.:

```
memcpy(data, "Hello world", 11);
```

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
0 1 2 3 4
```

[9] `https://learn.microsoft.com/en-us/windows/win32/debug/system-error-codes`

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.

Využijte kód z minulého cvičení.

8.2 Sdílená paměť

V případě operačního systému Windows můžeme mapování souborů do paměti využít k vytvoření sdílené paměti a komunikaci mezi procesy, jako jsme to viděli v minulém cvičení. V přiložených zdrojových kódech je demonstrována komunikace dvou procesů, kde jeden předává data do sdílené schránky a druhý tato data čte.

8.2.1 Sdílená paměť jako soubor namapovaný do paměti

Při vytváření sdílené paměti mezi těmito procesy nejdříve jeden z nich (v našem případě zapisující proces) vytvoří objekt mapování do paměti a inicializuje tuto oblast. Tento objekt mapování odpovídá mapování souboru do paměti v režimu pro čtení a zápis, avšak je tomuto objektu (posledním argumentem) přiřazeno pojmenování.

```
TCHAR* mboxName = _T("mbox");
hMapping = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, sizeof(struct mbox), mboxName);
```

Takto vytvořený objekt může získat druhý proces pomocí funkce `OpenFileMapping`.[10]

```
hMapping = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, mboxName);
```

A následně provést namapování pomocí `MapViewOfFile`.

Analogickým způsobem můžeme sdílet synchronizační prostředky mezi procesy. Jeden proces (v našem případě opět zapisující proces) vytvoří zámek, který bude synchronizovat přístup do sdílené paměti.

```
TCHAR* mboxLockName = _T("mboxlock");
hLock = CreateMutex(NULL, FALSE, mboxLockName);
```

A druhý proces k tomuto zámku získá přístup pomocí funkce `OpenMutex`[11].

[10] `https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-openfilemappinga`

[11] `https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-openmutexw`

```
hLock = OpenMutex(SYNCHRONIZE, FALSE, mboxLockName);
```

Poznámka: Zde vidíme, jak se ze standardních synchronizačních nástrojů, které jsme dosud používali pouze pro vlákna, mohou stát nástroje pro synchronizaci procesů.

8.2.2 Anonymní sdílená paměť

V předchozím případě jsme pro vytvoření sdílené paměti použili soubor. To nemusí být vždy žádoucí. Pokud chceme namapovat paměť, která není svázana s žádným běžným souborem,[12] můžeme funkci `CreateFileMapping` předat místo handle otevřeného souboru hodnotu `INVALID_HANDLE_VALUE`. V takovém případě dojde k namapování stránek ze stránkovacího souboru.[13]

Tuto paměť můžeme sdílet mezi procesy, protože k její identifikaci slouží pojmenování objektu, které uvedeme u funkce `CreateFileMapping`. Ukázkový příklad můžeme upravit následovně.

```
TCHAR* mboxName = _T("mbox");
hMapping = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0,
                             sizeof(struct mbox), mboxName);
```

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.

8.3 Ochrana paměti

Podobně jako v unixových operačních systémech můžeme využít vlastností mapování souboru do paměti k tomu, abychom vytvořili oblasti paměti, které mají specifické vlastnosti z pohledu ochrany přístupu, např. jsou jen pro čtení, umožňují spouštět kód apod.

Následující kód ukazuje, jak vytvořit oblast paměti, kde je uložený spustitelný kód. Konkrétně se jedná o funkci, která vrací hodnotu svého argumentu zvýšenou o jedna. Tato funkce v assembleru a strojovém kódu vypadá následovně:

```
0: 89 c8                    mov    eax,ecx
2: ff c0                    inc    eax
4: c3                       ret

typedef int (*intfun)(int);
char data[] = { 0x89, 0xc8, 0xff, 0xc0, 0xc3 };
int _tmain(){
    int length = sizeof(data);
    HANDLE hMapping = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_EXECUTE_READWRITE,
                                        0, length, NULL);
```

[12] Tj. ekvivalent mapování anonymní paměti v unixech.

[13] `pagefile.sys`

```
    if (hMapping == NULL) /* ... */
    LPVOID exData = MapViewOfFile(hMapping, FILE_MAP_ALL_ACCESS | FILE_MAP_EXECUTE,
                                  0, 0, length);
    memcpy(exData, data, length);
    intfun f = (intfun)exData;
    _tprintf(_T("%i\n"), f(10));
    CloseHandle(hMapping);
}
```

9

Souborový systém FAT32

Souborové systémy tvoří důležitou část operačních systémů. Ve srovnání s jinými částmi operačního systému máme v případě souborových systémů tu výhodu, že si jejich implementaci můžeme vyzkoušet pohodlně v uživatelském prostoru a v libovolném vhodném programovacím jazyce.[1] Náročnost implementace souborového systému se liší v závislosti na poskytovaných funkcích a použitých optimalizacích. V tomto cvičení si ukážeme základní strukturu a práci se souborovým systémem FAT32. Tento souborový systém je zajímavý jednak tím, že se s ním dá stále setkat na řadě míst, a hlavně tím, že je opravdu jednoduchý, jak by mělo být vidět z následujích řádek.[2]

9.1 Struktura souborového systému

Existuje několik verzí souborového systému FAT, které jsou označovány jako FAT12, FAT16 a FAT32.[3] Tyto verze mají identickou strukturu souborového systému, ale liší se formátem dat, zejména velikostí záznamů v tabulce FAT.[4] Obecnou strukturu souborového systému ukazuje následující schéma.[5]

rezervovaná oblast	tabulka FAT	kořenový adresář	oblast s daty
(boot sektor, informace FS)	(může být i kopie)	FAT12, FAT16 (u FAT32 možnost)	...

Která verze souborového systému bude použita, určuje velikost disku,[6] kde bude souborový systém uložen, přesněji řečeno, počet datových clusterů. Pro malé disky (s méně než 4085 clustery) se použije FAT12,

[1] Protože se jazyk C běžně používá pro vývoj operačních systémů, bude to naše první volba, ale klidně bychom mohli použít jazyk Java nebo Python.

[2] První verze souborového systému FAT se používaly na počítačích, které měly řádově *desítky* (maximálně nižší stovky) *kilobytů* RAM a podobně jako zbytek operačního systému MS-DOS byl souborový systém kompletně napsán v assembleru procesoru 8086. To je také důvod, proč souborový systém FAT obsahuje jen ty opravdu nejnutnější věci a z dnešního pohledu má řadu vážných nedostatků.

[3] Ještě existuje nástavba označovaná jako Virtual FAT (nebo VFAT), která je kompatibilní se souborovým systémem FAT a přidává podporu dlouhých jmen souborů.

[4] Budeme rozlišovat *souborový systém FAT* (jako celek) a *tabulku FAT* (strukturu nesoucí informace o umístění souborů).

[5] Více informací např. `https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system` nebo `https://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc`

[6] V tomto textu budeme používat pojem disk, i když souborový systém může být umístěn i jinde, např. v oddílu disku.

pro větší disky (s méně než 65525 clustery) se použije FAT16 a pro (zbývající) velké disky se použije FAT32. Z toho plyne, že pokud bychom chtěli implementovat opravdu plnohodnotný souborový systém FAT, museli bychom implementovat všechny tři verze současně. V následujícím textu si práci usnadníme a budeme implementovat jen verzi FAT32 a uvedené informace se budou vztahovat pouze k této verzi. Implementace zbývajících verzí by byla velmi podobná, avšak místy bychom mohli narazit na drobné odlišnosti.

9.1.1 Boot sektor

První sektor disku obsahuje (mimo jiné) podrobné informace o vlastnostech souborového systému. Nás budou zajímat ty, které přesně definují jeho strukturu, např. velikost clusteru. Tabulka 9.1 představuje informace, které budeme potřebovat pro čtení ze souborového systému a pochází (včetně značení) z oficiální dokumentace.[7]

označení	offset (B)	velikost (B)	popis
`BPB_BytsPerSec`	11	2	velikost sektoru (v bytech); typicky 512
`BPB_SecPerClus`	13	1	počet sektorů v jednom clusteru (musí být mocnina 2)
`BPB_RsvdSecCnt`	14	2	počet sektorů rezervovaných na začátku disku
`BPB_NumFATs`	16	1	počet FAT tabulek (typicky 2)
`BPB_TotSec32`	32	4	celkový počet sektorů na disku
`BPB_FATSz32`	36	4	počet sektorů, které zabírá jedna FAT tabulka
`BPB_ExtFlags`	40	2	příznaky udávající způsob práce s kopiemi FAT tabulky
`BPB_FSVer`	42	2	upřesnění verze souborového systému
`BPB_RootClus`	44	4	číslo prvního clusteru kořenového adresáře
`BPB_FSInfo`	48	2	číslo sektoru v rezervované oblasti, kde jsou uloženy další informace souborového systému
`BS_BootSig`	66	1	příznak, že další 3 informace jsou přítomny
`BS_VolID`	67	4	pseudo-unikátní identifikátor média
`BS_VolLab`	71	11	popisek disku
`BS_FilSysType`	82	8	vždy obsahuje řetězec `"FAT32   "`

Tabulka 9.1: Vybrané informace uložené v bootsectoru

9.1.2 Adresáře

Metadata jednotlivých souborů (včetně adresářů) jsou uložena jako záznamy v adresářích. Tyto záznamy mají pevnou velikost 32 B a jejich strukturu popisuje Tabulka 9.2.

Názvy souboru jsou vždy ve tvaru 8.3, tj. 8 znaků jméno, 3 znaky přípona. Pokud je jméno nebo přípona souboru kratší, je volné místo doplněno mezarami (znak `0x20`), tečka není uložena, nerozlišují se malá a velká písmena a jméno souboru musí obsahovat minimálně jeden znak jiný než mezera. Pokud je první byte názvu souboru:

- znak `0xe5`, jedná se o neplatný záznam (např. smazaný soubor).

[7] `https://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc`

označení	offset (B)	velikost (B)	popis
`name`	0	11	název souboru
`attr`	11	1	atributy souboru
`ntRes`	12	1	rezervováno pro Windows NT
`crtTimeTenth`	13	1	čas vytvoření souboru
`fstClustHI`	20	2	číslo prvního clusteru souboru (horních 16 bitů)
`wrtTime`	22	2	čas posledního zápisu do souboru
`wrtDate`	24	2	datum posledního zápisu do souboru
`fstClustLO`	26	2	číslo prvního clusteru souboru (spodních 16 bitů)
`fileSize`	28	4	velikost souboru (v bytech)

Tabulka 9.2: Vybrané informace uložené v bootsectoru

- znak `0x00`, jedná se o neplatný záznam a příznak, že další zaznamy v adresáři jsou taky neplatné. (Není tak nutné procházet další záznamy.)

Atributy souboru jsou následující:

```
ATTR_READ_ONLY    0x01
ATTR_HIDDEN       0x02
ATTR_SYSTEM       0x04
ATTR_VOLUME_ID    0x08
ATTR_DIRECTORY    0x10
ATTR_ARCHIVE      0x20
ATTR_LONG_NAME    ATTR_READ_ONLY | ATTR_HIDDEN | ATTR_SYSTEM | ATTR_VOLUME_ID
```

9.1.3 FAT tabulka

V adresáři máme u každého souboru uloženu informaci, kde se nachází první cluster souboru. Abychom mohli najít další clustery souboru, obsahuje souborový systém tabulku FAT, která pro daný cluster souboru ukazuje na následující cluster, pokud existuje. Pokud takový cluster neexistuje, je tam uložen příznak EOC (end of chain), což je hodnota větší nebo rovna `0x0FFFFFF8`. Z pohledu implementace můžeme na tabulku FAT nahlížet jako na pole 32bitových hodnot.

Pozor: První datový cluster je cluster s číslem 2.

9.2 Implementace

Známe-li strukturu souborového systému, vytvoření kódu, který s tímto souborovým systém pracuje, je přímočaré. Jelikož naše implementace bude postavena čistě na kódu v uživatelském prostoru, potřebujeme nějakým způsobem emulovat pevný disk. Zde se nabízí použít běžný soubor, se kterým budeme pracovat jako s blokovým zařízením, tj. budeme číst (popř. zapisovat) vždy celé sektory, např. 512 B.

9.2.1 Strukturované datové typy

Informace, které jsme si popsali v úvodní kapitole, budeme potřebovat uložit do vhodných datových typů. Pro reprezentaci souborového systému zavedeme typ `struct fat32`, který kromě výše popsaných informací obsahuje odkaz na soubor (image), kde je daný souborový systém uložen a další praktické informace, jmenovitě:

- první datový sektor (`FirstDataSector`),
- počet datových sektorů (`DataSec`),
- počet datových clusterů (`CountOfClusters`),
- velikost cluster (`ClusterSz`).

Kód této struktury vypadá následovně a najdeme jej v souboru `fat32.h`.

```
struct fat32 {
        FILE    *image;
        // data ulozena na disku
        uint16_t BPB_BytsPerSec;
        uint8_t  BPB_SecPerClus;
        uint16_t BPB_RsvdSecCnt;
        uint8_t  BPB_NumFATs;
        uint32_t BPB_TotSec32;
        uint32_t BPB_FATSz32;
        uint16_t BPB_ExtFlags;
        uint16_t BPB_FSVer;
        uint32_t BPB_RootClus;
        uint16_t BPB_FSInfo;
        uint8_t  BS_BootSig;
        uint32_t BS_VolID;
        char     BS_VolLab[11];
        char     BS_FilSysType[8];
        // data odvozena
        uint32_t ClusterSz;
        uint32_t FirstDataSector;
        uint32_t DataSec;
        uint32_t CountOfClusters;
};
```

V podobném duchu si zavedeme strukturovaný datový typ `struct fat32_dir_entry` (viz následující kód) představující záznamy v adresářích. Abychom si práci zjednodušili, informace o čase budeme ignorovat. A dále si zavedeme atribut, který sloučí do jedné hodnoty pozici prvního clusteru v souboru.

```
struct fat32_dir_entry {
        // data ulozena na disku
        char     fileName[11];
        uint8_t  attr;
        uint16_t fstClusterHI;
        uint16_t fstClusterLO;
        uint32_t fileSize;
        // data odvozena
        uint32_t fstClusterId;
};
```

Vedle těchto dvou datových typů si zavedeme ještě strukturovaný datový typ `struct fat32_fd`, který bude reprezentovat otevřený soubor (resp. adresář) a který má následující atributy.

```
struct fat32_fd {
    struct fat32 *fs;        // souborovy system
    uint32_t fileOffset;     // aktualni pozice v souboru
    uint32_t fileSize;       // velikost souboru
    uint32_t currentCluster; // aktualni cluster
    uint32_t bufOffset;      // aktualni pozice v bufferu
    uint32_t bufSize;        // pocet platnych bytu v bufferu
    int      isDirectory;
    uint8_t  buf[];          // buffer s nactenymi daty
};
```

9.2.2 Pomocné funkce

Nejdříve si zavedeme pomocnou funkci, která nám umožní pracovat se souborem jako s blokovým zařízením. Funkce `sector_read` slouží k tomu, abychom ze souborového systému `fs` načetli sektor `sector` a uložili jej do bufferu `buf`. Aby tato funkce byla obecnější, obsahuje ještě parametr `count`, který udává, kolik sektorů za sebou se má načíst.

```
static void sector_read(struct fat32 *fs, uint32_t sector, uint32_t count, uint8_t *buf) {
    fseek(fs->image, (long) sector * fs->BPB_BytsPerSec, SEEK_SET);
    fread(buf, fs->BPB_BytsPerSec, count, fs->image);
}
```

Tuto pomocnou funkci využijeme k vytvoření další pomocné funkce a to jest funkce, která přečte zadaný cluster:

```
static void cluster_read(struct fat32 *fs, uint32_t clusterId, uint8_t *buf) {
    uint32_t sector = ((clusterId - 2) * fs->BPB_SecPerClus) + fs->FirstDataSector;
    sector_read(fs, sector, fs->BPB_SecPerClus, buf);
}
```

9.2.3 Inicializace souborového systému

Než můžeme začít pracovat se soubory v souborovém systému, musíme provést inicializaci odpovídajících datových struktur. Postupujeme tak, že přečteme první sektor a z boot sectoru přečteme jednotlivé hodnoty, např. následovně.

```
void fat32_init(struct fat32 *fs, FILE *image) {
    uint8_t sector[512];
    fread(sector, 1, 512, image);
    fs->image = image;
    memcpy(&fs->BPB_BytsPerSec, sector + 11, 2);
    memcpy(&fs->BPB_SecPerClus, sector + 13, 1);
    // ...
```

Poznámka: V tomto případě jsme si (pro přehlednost) kód zjednodušili. Vícebytové hodnoty jsou v případě souborového systému uloženy ve formátu little-endian, což je stejný formát, jaký používají procesory x86, není proto potřeba konverze a můžeme použít funkci `memcpy`. Pokud bychom chtěli náš kód použít na procesorech pracujích s hodnotami big-endian, museli bychom provést odpovídající konverzi.

9.2.4 Otevření a čtení souboru

Pro otevření souboru si vytvoříme funkci `fat32_open`, která inicializuje strukturu `struct fat32_fd`, tj.

1. alokuje tuto strukturu,
2. alokuje prostor pro buffer, který má velikost jednoho clusteru,
3. nastaví výchozí hodnoty,
4. načte do bufferu první cluster souboru, pokud existuje.

Při čtení obsahu souboru (viz funkce `fat32_read` v souboru `fat32.c`) postupujeme tak, že čteme data z bufferu, který je součástí `struct fat32_fd`, a nejsou-li v tomto bufferu další data k dispozici, načteme do něj obsah dalšího clusteru, viz funkce `fat32_next_cluster`. K zjištění, kde se nachází další cluster souboru, slouží funkce `fat32_next_cluster_id`:

```
1 static uint32_t fat32_next_cluster_id(struct fat32 *fs, uint32_t currentCluster) {
2     uint8_t  buf[fs->BPB_BytsPerSec];
3     uint32_t fat_offset = currentCluster * 4;
4     uint32_t fat_sector = fs->BPB_RsvdSecCnt + (fat_offset / fs->BPB_BytsPerSec);
5     uint32_t fat_entry  = fat_offset % fs->BPB_BytsPerSec;
6     sector_read(fs, fat_sector, 1, buf);
7     uint32_t rawId = *((uint32_t *) (buf + fat_entry));
8     return rawId & 0x0fffffff;
9 }
```

Tato funkce přímo pracuje s tabulkou FAT. Připomeňme, že se jedná o pole (uložené na disku), kde každá jedna (32bitová) položka obsahuje číslo následujícího clusteru.

Nejdříve určíme offset položky (v bytech), která nás zajímá (hodnota `fat_offset`). Z něj odvodíme sektor, kde se tento záznam nachází (hodnota `fat_sector`) a odpovídající byte v rámci tohoto sektoru (hodnota `fat_entry`).

Nyní můžeme sektor přečíst (řádek 6) a přečíst i odpovídající záznam (řádek 7) a to tak, že přetypujeme pole bytů na pole 32bitových hodnot. Protože podle specifikace cluster může být jen 28bitová hodnota, ještě provedeme zúžení na 28bitů (řádek 8).

9.2.5 Čtení obsahu adresáře

Při čtení obsahu adresáře postupujeme analogicky jako v případě čtení obsahu souboru, tj. postupně čteme 32bytové položky z bufferu, který je součástí struktury `struct fat32_fd`, viz funkce `fat32_read_dir`. Pokud tato funkce narazí na platný záznam, vrátí kladnou hodnotu a naplní zadanou strukturu `struct fat32_dir_entry`. Pokud v adresáři nejsou žádné další platné záznamy, vrátí funkce hodnotu `0`. Všimněme si, že tato funkce funguje de facto jako iterátor nad obsahem adresáře.

9.3 Ukázková aplikace

Součástí přiložených zdrojových kódů je i aplikace umožňující zobrazit metadata souborového systému i jednotlivých souborů, a taky zobrazit obsah souboru. Pro vyzkoušení je přiložen i souborový systém, který byl vytvořený pomocí příkazů:

```
dd if=/dev/zero of=tutorial09.fat32 bs=4096 count=32768
/sbin/mkfs.fat -F 32 -s 2 tutorial09.fat32
```

Úkol č. 1: Vyzkoušejte si, jak se změní parametry souborového systému, pokud zvolíme jinou velikost (viz příkaz `dd`) nebo jiný počet sektorů na cluster (přepínač `-s`).

Operační systém Linux umožňuje připojit takto vytvořený souborový systém jako běžný adresář pomocí příkazu:

```
mount -o loop tutorial09.fat32 /mnt/foo
```

Provedení tohoto příkazu však vyžaduje administrátorská oprávnění.

Úkol č. 2: Vytvořte si nástroj, který vám umožní přečíst všechny soubory (i ve vnořených adresářích) v souborovém systému FAT32 a uložit je na (jiný) disk.

Úkol č. 3: Použijte tento nástroj.

10

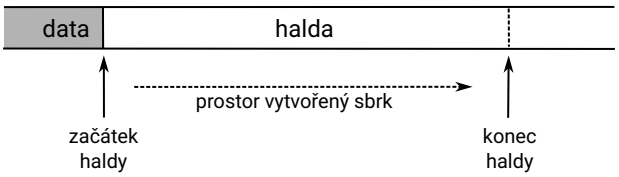
Alokace paměti na haldě (Linux)

Programy ke své práci obvykle využívají dvě oblasti paměti – *zásobník* (stack) a *haldu* (heap). Zatímco zásobník obvykle slouží k uložení lokálních proměnných, jejichž platnost je pouze po dobu provádění dané funkce, halda slouží k dynamické alokaci objektů (úseků paměti), které můžeme použít i po skončení funkce, která je alokovala. V tomto cvičení si ukážeme základní principy, které se používají při dynamické alokaci paměti.

10.1 Získání oblasti paměti

Práci s haldou zajišťuje buď standardní knihovna (např. jazyk C) nebo běhové prostředí (např. jazyk Java, C#). V obou případech běžící proces získá od operačního systému oblast paměti, kterou podle potřeby dělí na menší úseky (objekty). V jazyce C k získání těchto úseků slouží funkce `malloc` a k uvolnění již nepoužívaný úseků paměti slouží funkce `free`.

K získání oblasti paměti, kterou budeme dělit na menší části, můžeme v unixových operačních systémech použít systémové volání `sbrk`. Toto systémové volání rozšíří datový segment procesu[1] o zadaný počet bytů. Návratovou hodnotou je ukazatel na předchozí konec datového segmentu. Použití `sbrk` ilustruje Obrázek 10.1.



Obrázek 10.1: Halda

Pro správu této oblasti paměti si zavedeme několik globálních proměnných.

```
unsigned char *data_area = NULL;       // ukazatel na volnou oblast haldy
unsigned char *data_area_start = NULL; // ukazatel na zacatek haldy
```

[1] Ve smyslu oblasti paměti, kterou je možné použít pro práci s daty, nemusí se jednat o datový segment ve smyslu procesorů x86.

```
size_t data_area_capacity = 0;          // velikost haldy v bytech
```

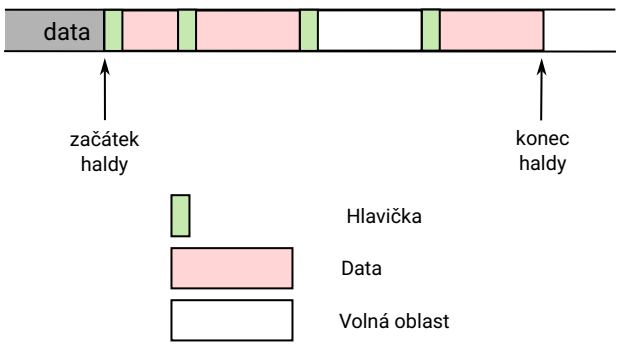
A funkci, která v případě potřeby inicializuje oblast, kde bude halda, případně ji rozšíří.

```
/** rozsiri haldu o MIN_ACQ_SIZE */
static void acquire_mem()
{
        unsigned char *last_pos = sbrk(MIN_ACQ_SIZE);
        if (!data_area_start) {
                data_area_start = last_pos;
                data_area = data_area_start;
        }
        data_area_capacity += MIN_ACQ_SIZE;
}
```

Poznámka: Alternativně můžeme k získání oblasti paměti, kterou je možné dále dělit, použít mapování souboru do paměti. Buď můžeme namapovat do paměti soubor `/dev/zero` nebo použít mapování anonymní oblasti paměti.

10.2 Struktura haldy

Máme-li k dispozici souvislou oblast paměti, můžeme ji dělit na menší bloky. Každý takový blok se skládá z hlavičky nesoucí informace o jednotlivých blocích (např. jeho velikost) a z části obsahující data, viz Obrázek 10.2. Bloky dále budeme dělit na *používané*, obsahující platná data, a *uvolněné*, které je možné opětovně použít. V našem případě budeme předpokládat, že každý objekt má hlavičku složenou ze dvou slov a pro data je alokována paměť minimálně o velikosti dvou slov.[2]



Obrázek 10.2: Ilustrace rozdělení haldy na menší bloky

K popisu obou typů objektů si zavedeme strukturovaný datový typ:

```
struct chunk {
    size_t prev_size;    // velikost predchoziho bloku (v bytech, vcetne hlavicky)
    size_t size;         // velikost bloku (v bytech, vcetne hlavicky)
```

[2]Tj. při každé alokaci je použita oblast minimálně o velikosti čtyř slov.

```
    struct chunk *prev;  // odkaz na predchozi volny blok
    struct chunk *next;  // odkaz na dalsi volny blok
};
```

V této struktuře první dva atributy představují hlavičku bloku paměti, kdy `size` udává velikost celého bloku v bytech a to včetně hlavičky, a `prev_size` udává velikost předchozího bloku[3] a v nejnižším bitu tohoto atributu si udržujeme informaci o tom, zda je aktuální blok používaný (bit je nastaven na 0) nebo je uvolněný (bit je nastaven na 1).[4] Atributy `prev` a `next` jsou již v datové části bloku a slouží k vytvoření spojového seznamu volných bloků.

10.2.1 Práce s jednotlivými bloky

Pro manipulaci s jednotlivými bloky dat si zavedeme několik pomocných funkcí.

Následující funkce pro zadaný blok vrací odkaz na jeho datovou část (oblast za hlavičkou). V tomto kódu se používá přetypování na typ `unsigned char *`, díky čemuž můžeme (společně s pointerovou aritmetikou) dohledat příslušný byte v paměti, tj. posunout se za hlavičku daného bloku.

```
static inline void *chunk_to_ptr(struct chunk *mem) {
    return ((unsigned char *) mem) + HDR_SIZE;
}
```

Analogicky můžeme z ukazatele na datovou oblast získat odkaz na celý blok včetně hlavičky.[5]

```
static inline struct chunk *ptr_to_chunk(void *ptr) {
    return (struct chunk *) (((unsigned char *) ptr) - HDR_SIZE);
}
```

Dále si zavedeme funkce, které umožní zjistit a nastavit příznak, zda je blok používaný nebo uvolněný.

```
static inline int chunk_is_free(struct chunk *mem) {
    return mem->prev_size & 0x01;
}
static inline void chunk_set_free(struct chunk *mem, int free) {
    mem->prev_size = (mem->prev_size & ~0x01) | free;
}
```

K procházení bloků paměti na haldě si vytvoříme další dvě pomocné funkce, které vrátí odkaz na předchozí nebo následující blok na haldě, pokud takové bloky existují.

[3] Tuto informaci můžeme využít při slučování sousedních volných bloků.

[4] To můžeme bezpečně použít, protože velikost bloků paměti jsou zaokrouhleny na dvojnásobky slov, tudíž jsou nejnižší bity vždy 0.

[5] Protože jsou obě funkce relativně malé, jsou deklarovány jako `inline`. Překladač pak nebude volat funkce tradičním způsobem, ale vloží jejich tělo na místo v programu, kde jsou použity. V případě funkce `chunk_to_ptr` nemusíme explicitně uvádět převod na typ `void *`, ten se v jazyce C provádí implicitně.

```
static inline struct chunk *chunk_preceding(struct chunk *mem) {
    unsigned char *pos = (unsigned char *) mem;
    if (pos == data_area_start) return NULL;
    return (struct chunk *) (pos - (mem->prev_size & ~0x01));
}
static inline struct chunk *chunk_succeeding(struct chunk *mem) {
    unsigned char *pos = (unsigned char *) mem;
    if ((pos + mem->size) == data_area) return NULL;
    return (struct chunk *) (pos + mem->size);
}
```

10.2.2 Alokace bloků

Při alokaci bloků paměti máme dvě možnosti, jak postupovat. Buď můžeme „recyklovat" již nepoužíváný blok nebo alokovat blok nový. Tento princip se dá přímo popsat následujícím kódem.

```
void *tmalloc(size_t size) {
        size = ROUND_UP(size + HDR_SIZE);
        struct chunk *mem = chunk_find(size);
        if (mem) chunk_set_free(mem, 0);
        else mem = chunk_allocate(size);
        return chunk_to_ptr(mem);
}
```

Nejdříve spočítáme velikost bloku, který potřebujeme (přičteme velikost hlavičky a zaokrouhlíme na dvojnásobek velikosti slova). Funkcí `chunk_find` zkusíme najít vhodný uvolněný blok, a pokud existuje, označíme jej jako používaný a použijeme jej. V opačném případě alokujeme blok nový. Funkce `tmalloc`, která je ekvivalentem `malloc`, vrátí odkaz na datovou část bloku bez ohledu na to, jestli se jedná o nový nebo „recyklovaný" blok.

Podívejme se nejprve na to, jak alokovat nový blok.

```
1  static struct chunk *chunk_allocate(size_t size) {
2      while (size > data_area_capacity)
3          acquire_mem();
4      struct chunk *mem = (struct chunk *) data_area;
5      data_area += size;
6      data_area_capacity -= size;
7      mem->size = size;
8      if (last_chunk) mem->prev_size = last_chunk->size;
9      else mem->prev_size = 0;
10     last_chunk = mem;
11     return mem;
12 }
```

Nejdříve zajistíme, že máme na haldě dostatek místa, které můžeme přidělit (řádky 2 a 3). Alokujeme příslušný blok (řádky 4 až 6) nastavíme hodnoty v hlavičce (řádky 7 až 9). Abychom věděli, jak velký byl předchozí blok, použijeme globální proměnnou `last_chunk`, která obsahuje ukazatel na poslední alokovaný blok na haldě (pokud takový existuje).

10.2.3 Uvolnění bloku

Abychom mohli opětovně použít uvolněné bloky, musíme mít o nich přehled. Pro jednoduchost budeme všechny uvolněné bloky shromažďovat v oboustranném spojovém seznamu, který je určen globální proměnnou `free_blocks`. Práci s tímto seznamem zajišťuje dvojice funkcí `chunk_enqueue` a `chunk_dequeue`, které daný blok zařadí na seznam nebo z něj vyjmou, viz přiložené zdrojové kódy.

Při uvolňování bloku postupujeme tak, že se podíváme na sousední bloky, a pokud jsou volné, sloučíme je s uvolňovaným blokem, jak ilustruje následující kód.

```
1  static void chunk_enqueue_merged(struct chunk *mem) {
2      struct chunk *prec = chunk_preceding(mem);
3      struct chunk *succ = chunk_succeeding(mem);
4      if (prec && chunk_is_free(prec)) {
5          chunk_dequeue(prec);
6          prec->size += mem->size;
7          mem = prec;
8      }
9      if (succ && chunk_is_free(succ)) {
10         chunk_dequeue(succ);
11         mem->size += succ->size;
12         succ = chunk_succeeding(mem);
13     }
14     if (succ) {
15         succ->prev_size = mem->size;
16         chunk_enqueue(mem);
17     } else {
18         data_area -= mem->size;
19         data_area_capacity += mem->size;
20         last_chunk = chunk_preceding(mem);
21     }
22 }
```

Pokud je předchozí blok volný (řádky 4 až 8), je tento blok zvětšen o námi uvolněný blok a dále pracujeme s tímto (předchozím) blokem. Pokud je následující blok volný, je uvolňovaný blok rozšířen a jeho velikost. V obou případech jsou sousední volné bloky odstraněny ze seznamu uvolněných bloků. Díky tomuto přístupu, nejsou v paměti nikdy dva volné bloky vedle sebe.

V případě, že existuje za uvolněným blokem blok paměti, který je používaný, je mu upravena hlavička a uvolněný blok je zařazen na seznam uvolněných bloků (řádky 14 až 16), v případě, že jsme uvolnili poslední blok na haldě, můžeme oblast haldy zmenšit (řádky 18 až 20).

S pomocnou funkcí `chunk_enqueue_merged` je implementace ekvivalentu funkce `free` přímočará.

```
void tfree(void *ptr) {
    struct chunk *mem = ptr_to_chunk(ptr);
    chunk_enqueue_merged(mem);
}
```

10.2.4 Vyhledání uvolněného bloku

U funkce `tmalloc` jsme předpokládali, že existuje funkce `chunk_find`, která najde uvolněný blok vhodné velikosti. Nyní, když víme, jak jsou uvolněné bloky organizovány, můžeme tuto funkci doplnit. Následující funkce je jedna z možných a implementuje strategii *first-fit*, tj. je použit první vhodný blok.

```
static struct chunk *chunk_find(size_t size) {
    if (!free_blocks) return NULL;
    struct chunk *c = free_blocks;
    while (c) {
        if (c->size >= size) {
            chunk_dequeue(c);
            return c;
        }
        c = c->next;
    }
    return NULL;
}
```

Poznámka: V tomto bodě je již dynamická alokace kompletní, i když má četné nedostatky. Zejména správa volných bloků a jejich vyhledávání není efektivní. Důležité je i zmínit, že takto naprogramovanou alokaci paměti nemůžeme použít ve vícevláknových aplikacích. Pro ně by bylo nutné doplnit zamykaní.

10.3 Úkoly

1. Struktura haldy je zvolena tak, aby ji šlo snadno analyzovat, k tomu slouží funkce `tmalloc_debug`. Podívejte se, jak tato funkce funguje, a vyzkoušejte si alokovat/uvolňovat různé objekty.
2. Vytvořte (jednoduchý) spojový seznam a do něj načtěte různě dlouhé řádky ze souboru. Podívejte se, jak budou v paměti uloženy a jak pro ně bude alokován prostor.
3. Dvojici funkcí `tmalloc` a `tfree` doplňte o funkci `trealloc`, která se bude chovat jako funkce `realloc` ze standardní knihovny.
4. Vyzkoušejte funkci `trealloc` a podívejte se, jak je alokován a uvolňován prostor.
5. Upravte funkci `chunk_find`, aby používala strategii best-fit.
6. (**bonusový úkol**) Upravte práci s volnými bloky tak, aby bloky do určité velikosti, např. 512 B byly rozděleny do samostatných seznamů a nemusely být vyhledávany mezi ostatními (velkými) bloky.