Přenos dat a počítačové sítě

Projekt 1

Obsah

Zabezpečovací linkové kódy pro přenos dat - CRC 1

Zadání 1

Text zadání 1

Použité prostředky 1

Forma odevzdání 1

Literatura 1

Řešení blokového zpracování 2

Kódování 2

Dekódování 2

Obsah zakódovaného souboru 2

Algoritmus CRC 3

Popis 3

Implementace 3

Detekce chyby 3

Literatura 4

Zabezpečovací linkové kódy pro přenos dat - CRC

Zadání

Text zadání

Cílem projektu je vyzkoušet si kódování a dekódování dat zabezpečených CRC linkovým kódem.

Zabezpečte vstupní 64-bitové bloky dat níže uvedeným cyklickým součtem. Při dekódování detekujte chybu. Popište v dokumentaci princip vámi použitého algoritmu, způsob detekující chybu a omezení detekčních schopností. Demonstrujte funkčnost na příkladu (dávka testuj).

CRC-32 (*IEEE 802*) - generující polynom:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1 \qquad (1)$$

Úkolem je vytvořit program se jménem *kodovani*, který načte vstupní data (ze standardního vstupu) a podle zadaných parametrů je zakóduje nebo dekóduje. Výsledek vytiskne na standardní výstup. Syntaxe spouštení programu:

```
kodovani (-c | -d)
```

kde -c znamená kódování a -d dekódování. Příklad použití:

```
kodovani -c < data.txt # kódování
kodovani -d < kod.txt  # dekódování
```

Vstupní data jsou tvořena posloupností libovolných binárních dat (jako testovací data použijte například *jpg*, *pdf*, *doc*, *txt*, *bmp* apod.). Jestliže vstupní data netvoří potřebný počet bitů, doplňte zbývající bity bloku nulovými bity. Doplněné bity nezapomeňte při dekódování odstraňovat.

Součástí odevzdaného projektu bude dávka testuj, která bude demonstrovat funkčnost vašeho řešení na vámi zvoleném vstupním souboru dat (kódování, dekódování).

Použité prostředky

Program vytvořte v jazyce C/C++ tak, aby byl přeložitelný na serveru EVA (eva.fit.vutbr.cz) nebo na SUNech (např. sun00.fit.vutbr.cz). V dokumentaci uveďte na kterém ze serverů je projekt přeložitelný a funkční.

Pozor: Nepřeložitelné projekty nebudou hodnoceny.

Forma odevzdání

- zip archív se jménem login.zip (např. xnovak00.zip).
- součástí archívu budou:
    - funkční Makefile
    - komentované zdrojové kódy programů (kodovani.c)
    - dokumentace v pdf, ps nebo ASCII - v češtině (dokumentace.pdf, dokumentace.ps, dokumentace.txt)
    - testovací skript testuj, který projekt přeloží a otestuje jeho funkčnost na příkladu
    - soubor testovacích dat pro dávku testuj

Dodržujte zadaná jména souborů! Dodržujte parametry spouštění projektu - implementujte projekt tak, aby jej bylo možné spouštět z dávkových souborů (žádné interakce s uživatelem).

Literatura

- přednášky z PDT
- poznámky ze cvičení PDT
- doplňkové texty ke cvičením

Řešení blokového zpracování

Výpočet cyklického kódu se provádí po 64bitových blocích, což znamená, že každých 64 bitů vstupních dat je rozšířeno na výstupu o 32 bitů CRC kódu. Pokud vstup není zarovnán na tuto hranici, je daný blok doplněn nulami a pak je spočítán jeho CRC kontrolní součet.

Kódování

Protože v zadání je požadavek na dekódování jen tolika dat, kolik bylo původně zakódováno, musíme přenášet taky velikost dat. Zjistit velikost je ale trocha problematické, protože vstup se čte ze standardního vstupu, kterého velikost nemůžeme zjistit a ukládat ho do paměti by omezilo použitelnost. Řešením těchto problémů je průběžně počítat zakódované znaky a po skončení vstupu uložit hlavičku na konec výstupu.

```
void koduj() {
    int vstup;                                  // vstupni znak
    int velikost;                               // skutecna velikost vstupu
    blok_in  = 8;                               // cteme 8B (64b) bloky
    blok_out = 12;                              // zapisujeme 12B (96b) bloky
    while ((vstup=getchar())!=EOF)              // cteme vstup az ho precteme
        pridej_znak( vstup );                   // ..a znaky pridavame do buff.
    velikost = pozice;                          // ted zname skutecnou velikost
    dopln_buffer_nulami();                      // zpracujeme posledni blok
    zapis_hlavicku( velikost );                 // vygen. hlavicku na konec dat
}
```

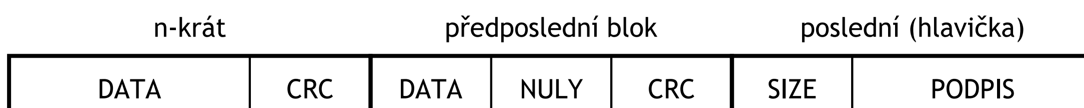
Dekódování

U dekódování se ověřuje CRC průběžně, ale výstup je zpožděný o jeden blok. Pokud se nalezne poslední blok zakódovaných dat, tak se zpracuje jak hlavička – tj. určí se kolik dat z předešlého bloku se reálně použije:

```
int dekoduj() {
    int zapsano = 0;
    int velikost;
    blok_in  = 12;                              // cteme 12B (96b) bloky
    blok_out = 0;
    while( nacti_buffer() ) {                   // cteme vstup
        if (konec_vstupu()) {                   // je to posledni blok?
            if (!dekoduj_hlavicku(&velikost))   // dekodujeme velikost
                return 0;                       // .. v pripade chyby konec
            blok_out = velikost - zapsano;      // toto schazi z predchoziho
            zapis_buffer2();                    // tak to zapiseme
            return 1;                           // a zde je uspesny konec
        } else {                                // neni to posledni blok
            zapsano += zapis_buffer2();         // spocteme zapsane byty
            if (!dekoduj_buffer())              // overime crc
                return 0;                       // .. v pripade chyby konec
            blok_out = 8;                       // mame nejake vystupni data
            zalohuj_buffer();                   // ktere zalohujeme
        }
    }
    return 0;                                   // blbe precteny blok
}
```

Velikost zakódovaného souboru je přesně 1,5 násobek velikosti původních dat (zaokrouhleno na 8 bajtů nahoru) + 12 bajtová hlavička.

Obsah zakódovaného souboru



Obr.1: Struktura zakódovaného souboru

Algoritmus CRC

Popis

Srozumitelný popis algoritmu jsem našel ve *Wikipedii* (viz [W]), kde byl uveden i pseudokód pro výpočet kontrolního součtu:

```
function crc(bit array bitString[1..len], int polynomial) {
     shiftRegister := initial value // commonly all 0 bits or all 1 bits
     for i from 1 to len {
         if most significant bit of shiftRegister xor bitString[i] = 1
             shiftRegister := (shiftregister left shift 1) xor polynomial
         else
             shiftRegister := shiftregister left shift 1
     }
     return shiftregister
 }
```

Implementace

Implementaci tohoto algoritmu jsem musel provést v programovacím jazyku C, který nepatří mezi moje oblíbené, takže jsem si dost často vypomáhal textem přednášek do kursu IJC. Nakonec se povedlo implementovat algoritmus jako:

```
// generujici polynom:
//    x^32+x^26 + x^23+x^22+x^16 + x^12+x^11+x^10+x^8 + x^7+x^5+x^4+x^2+x+1
// binarne:
//    1 00000100 11000001 00011101 10110111
//         0    4    C    1    1    D    B    7
#define POLYNOM 0x04C11DB7;

typedef unsigned int CRC32;                      // definice typu pro CRC

char    buffer[12];                              // buffer pro data

// makro ktere zjisti bit z bufferu ( b=xxxyyy kde x = bajt, y = bit v bajtu )
#define bufferbit(b) ( ( (buffer[b/8]) >> (b%8) ) & 1 )

//  Funkce ktera pocita crc z bloku
//
CRC32 spocti_crc() {
    CRC32 soucet = 0;                              // vychozi hodnota je nulova
    int i;                                         // bit
    for(i=0;i<64;i++) {
        if ( ( soucet >> 31 ) ^ bufferbit(i) ) {
            soucet = ( soucet << 1 ) ^ POLYNOM;
        } else {
            soucet = soucet << 1;
        }
    }
    return soucet;
}
```

Detekce chyby

Algoritmus CRC umí chybu jen detekovat, a tato detekce je v programu řešena porovnáním přijatého CRC a CRC spočteného z přijatých dat:

```
int dekoduj_buffer() {
    CRC32 *oldcrc;                               // stare crc - prenesene
    oldcrc = (CRC32*)(buffer+8);                 // stare crc je na 8.-11. byte
    if ( *oldcrc == spocti_crc() )               // detekce chyby,
        return 1;                                // je to ok
    fprintf(stderr,"Chyba: Vstupni data jsou poskozena!\n");
    return 0;                                    // neni to ok
}
```

Pokud nastala chyba při přenosu tak tyto dvě hodnoty CRC nebudou stejné.

Testovací dávka *testuj* provede dva testy a poškození souboru testuje jenom přes zvětšení souboru, ale ruční testování změnou jednoho písmene v zakódovaném souboru byla taky odhalena. Projekt byl bez problému otestován pomoc dávky *testuj* na serveru *eva.fit.vutbr.cz*.

Literatura

[W] **Cyclic redundancy check**
http://en.wikipedia.org/wiki/Cyclic_redundancy_check

[C] **Jazyk C – přednášky**
Dr. Ing. Petr Peringer
http://www.fit.vutbr.cz/study/courses/IJC/public/Prednasky/IJC.pdf

[BASH] **bash - GNU Bourne-Again SHell**
manuálová stránka programu bash(1), man bash