IJC: DU1

```
Jazyk C                     DU1                          20.2.2024
--------------------------------------------------------------------


                      Domácí úkol č.1


Termín odevzdání:  25.3.2024
 Hodnocení celkem max. 15 bodů
 Čtěte pokyny na konci tohoto textu

Příklady: (budou opravovány v prostředí Linux/GCC,
           LC_ALL=cs_CZ.utf8
           překlad: gcc -g -std=c11 -pedantic -Wall -Wextra
           C11 je potřeba jen pro static_assert)

A) V rozhraní "bitset.h" definujte datovou strukturu typu pole bitů:

   Typ:
     typedef <DOPLNIT> bitset_t;
       Typ bitového pole (pro předávání parametru do funkce odkazem).

     typedef unsigned long bitset_index_t;
        Typ indexu bitového pole.

   Makra:
     bitset_create(jmeno_pole,velikost)
       definuje a _nuluje_ proměnnou jmeno_pole
       (POZOR: opravdu musí _INICIALIZOVAT_ pole bez ohledu na
       to, zda je pole statické nebo automatické/lokální!
       Vyzkoušejte obě varianty, v programu použijte _lokální_ pole.)
       Použijte  static_assert  pro kontrolu velikosti pole.
       Př: static bitset_create(p,100); // p = pole 100 bitů, nulováno
           bitset_create(q,100000L);    // q = pole 100000 bitů, nulováno
           bitset_create(q,-100);       // chyba při překladu

     bitset_alloc(jmeno_pole,velikost)
       definuje proměnnou jmeno_pole tak, aby byla kompatibilní s polem
       vytvořeným pomocí bitset_create, ale pole bude alokováno dynamicky.
       Př: bitset_alloc(q,100000L); // q = pole 100000 bitů, nulováno
       Použijte  assert  pro kontrolu velikosti pole.
       Pokud alokace selže, ukončete program s chybovým hlášením:
       "bitset_alloc: Chyba alokace paměti"

     bitset_free(jmeno_pole)
       uvolní paměť dynamicky (pomocí bitset_alloc) alokovaného pole

     bitset_size(jmeno_pole)
       vrátí deklarovanou velikost pole v bitech (uloženou v poli)

     bitset_fill(jmeno_pole, bool_výraz)
       vynuluje(false) nebo nastaví na 1(true) celý obsah pole

     bitset_setbit(jmeno_pole,index,bool_výraz)
       nastaví zadaný bit v poli na hodnotu zadanou výrazem
       (nulový výraz == false == bit 0, jinak bit 1)
       Př: bitset_setbit(p,20,1);

     bitset_getbit(jmeno_pole,index)
       získá hodnotu zadaného bitu, vrací hodnotu 0 nebo 1
       Př: if(bitset_getbit(p,i)==1) printf("1");
           if(!bitset_getbit(p,i))   printf("0");
```

Kontrolujte meze polí. V případě chyby volejte funkci

```
error_exit("bitset_getbit: Index %lu mimo rozsah 0..%lu",
          (unsigned long)index, (unsigned long)mez).
```

(Můžete použít například modul error.c/error.h z příkladu b)

Programy musí fungovat na 32 (gcc -m32) i 64bitové platformě.

Podmíněným překladem zajistěte, aby se při definovaném symbolu
USE_INLINE místo těchto maker definovaly inline funkce stejného jména
všude kde je to možné (bez změn v následujícím testovacím příkladu!).
Pozor: USE_INLINE nesmí být definováno ve zdrojovém textu --
překládá se s argumentem -D (gcc -DUSE_INLINE ...).

Program musí fungovat s inline funkcemi i pro vypnuté optimalizace -O0
(ověřte si to - vyžaduje externí definice inline funkcí).

Pro vaši implementaci použijte pole typu unsigned long [].
V tomto poli na indexu 0 bude velikost bitového pole v bitech.
Implementace musí efektivně využívat paměť (využít každý
bit pole až na posledních maximálně CHAR_BIT*sizeof(unsigned long)-1 bitů).

Jako testovací příklad implementujte funkci, která použije algoritmus známý
jako Eratostenovo síto (void Eratosthenes(bitset_t pole);) a použijte ji
pro výpočet posledních 10 prvočísel ze všech prvočísel od 2 do
N=666000000 (666 milionů). Doporučuji program nejdříve odladit pro N=100.
Hodnotu N si funkce zjistí podle velikosti bitového pole.
Funkci Eratosthenes napište do samostatného modulu "eratosthenes.c".

Každé prvočíslo tiskněte na zvláštní řádek v pořadí
vzestupném. Netiskněte nic jiného než prvočísla (bude se
automaticky kontrolovat!). Pro kontrolu správnosti prvočísel
můžete použít program "factor" (./primes|factor).

Naprogramujte (s využitím funkce clock()) měření doby běhu programu v
sekundách a výsledek vypište na stderr následujícím příkazem:

```
fprintf(stderr, "Time=%.3g\n", (double)(clock()-start)/CLOCKS_PER_SEC);
```

(Porovnejte si vaše měření s výsledkem příkazu "time ./primes".)

Pro lokální pole budete potřebovat zvětšit limit velikosti zásobníku.
Na UNIX systémech můžete použít příkaz "ulimit -a" pro zjištění velikosti
limitu a potom "ulimit -s zadana_velikost_v_KiB" před spuštěním programu.
(Toto názorně demonstruje nevhodnost používání velkých lokálních polí.)

Zdrojový text programu se musí jmenovat "primes.c" !
Napište Makefile tak, aby příkaz "make" vytvořil všechny varianty:

```
primes      používá makra
primes-i    používá inline funkce
```

a aby příkaz "make run" všechny varianty vytvořil a spustil (i s ulimit -s).

(Při nesplnění výše uvedených podmínek: až 0 bodů.)

(8b)

Poznámky: Eratosthenovo síto (přibližná specifikace):

1) Nastavíme bitové pole p o rozměru N na samé 1.
   Nastavíme p[0]=0; p[1]=0; // 0 a 1 nejsou prvočísla
   index i nastavit na 2
2) Vybereme nejmenší index i, takový, že p[i]==1.
   Potom je i prvočíslo
3) Pro všechny násobky i nastavíme bit p[n*i] na 0
   ('vyškrtneme' všechny násobky i - nejsou to prvočísla)
4) i++; dokud nejsme za sqrt(N), opakujeme bod 2 až 4
   (POZOR: sestavit s matematickou knihovnou parametrem -lm)
5) Výsledek: v poli p jsou na prvočíselných indexech hodnoty 1,
   (=nebyly vyškrtnuty jako násobek nějakého menšího prvočísla)

https://en.wikipedia.org/wiki/Prime_number

```
   Efektivita výpočtu: cca 4.25s na Ryzen 5 4600G (gcc -O2)
   Porovnejte efektivitu obou variant (makra vs. inline funkce).
   Zamyslete se, jak by se ověřila efektivita pro (neinline) funkce.
```

```
B) Napište modul "error.c" s rozhraním v "error.h", který definuje
   funkci void warning(const char *fmt, ...) a
   funkci void error_exit(const char *fmt, ...).
   Tyto funkce mají stejné parametry jako printf(); tisknou
   text "Warning: " nebo "Error: " a potom chybové hlášení podle
   formátu fmt. Vše se tiskne do stderr (standardní funkcí vfprintf)
   a potom pouze error_exit ukončí program voláním funkce exit(1).
   Použijte definice ze <stdarg.h>.

   Napište program "no-comment.c", který vynechá poznámky ze zdrojového kódu
   jazyka C, který načtete ze souboru zadaného jako jediný argument programu.
   Pokud argument programu chybí, čte stdin. Výstup bude vždy na stdout
   (POZOR: pokud bude omylem přesměrován do vstupního souboru, chování je
   nedefinováno - tento problém lze zjistit pomocí POSIX funkce fstat).

   Musíte použít stavový automat a řešit i znaky v řetězcových a znakových
   literálech (např. řetězec "/***/" není poznámka = nenahrazovat mezerou,
   "text\"text", '\\' a '\''  musí fungovat, atd.).

   Program použije error_exit v případě chyby čtení souboru (soubor
   neexistuje, neukončená poznámka nebo řetězec atd.), jinak
   předpokládejte syntakticky korektní vstupní soubor (půjde přeložit)
   a zdrojové kódování ve formátu UTF-8 (neměl by být žádný problém
   se zpracováním po 8bit znacích, protože znaky /*'"\ jsou v ASCII).

   Použijte program "make" pro překlad/sestavení programu.
   Testovací příkaz:  ./no-comment no-comment.c
                      ./no-comment no-comment.c >no-comment-result

(7b)

   Zařiďte, aby příkaz "make" bez parametrů vytvořil všechny spustitelné
   soubory pro DU1.  Při změně kteréhokoli souboru musí přeložit jen změněný
   soubor a závislosti. Pokud bude Makefile vypadat jako skript, odečtou se 3b.
```

Testovací obrázek: du1-obrazek.ppm

```
C) Obecné pokyny pro vypracování domácích úkolů (rev 21.2.2024)

*  Pro úkoly v jazyce C používejte ISO C11 (soubory *.c)
   Použití nepřenositelných konstrukcí není dovoleno.

*  Úkoly zkontrolujte překladačem například takto:
      gcc -g -std=c11 -pedantic -Wall -Wextra priklad1.c
   místo gcc můžete použít i jiný překladač.
   V  souvislosti s tím napište do poznámky na začátku
   souboru jméno překladače, kterým byl program testován
   (implicitní je verze GNU C instalovaná na serveru merlin).

   POZOR: Zkontrolujte paměťové operace speciálním parametrem překladu.
   (Makefile: CFLAGS += -fsanitize=address, LDFLAGS += -fsanitize=address).

*  Programy  pište, pokud je to možné, do jednoho zdrojového
   souboru. Dodržujte předepsaná jména souborů.
```

```
*  Na začátek každého souboru napište poznámku, která bude
   obsahovat jméno, fakultu, označení příkladu a datum.

   Příklad:
   // primes.c
   // Řešení IJC-DU1, příklad a), 20.3.2111
   // Autor: Kapitán Nemo, FIT
   // Přeloženo: gcc 10.2
   // ...popis příkladu - poznámky, omezení, atd

* Úkoly je nutné zabalit programem zip takto:
       zip xnemok99.zip *.c *.h Makefile

  Jméno xnemok99 nahradíte vlastním. ZIP neobsahuje adresáře.
  Každý si zkontroluje obsah ZIP archivu jeho rozbalením v prázdném adresáři
  a napsáním "make run".

* Řešení se odevzdává elektronicky v ISVUT (velikost souboru je omezena)

* Odvezdávejte pouze nezbytně nutné soubory -- ne *.EXE !

* Úkoly neodevzdané v termínu budou za 0 bodů.

* Opsané úkoly budou hodnoceny 0 bodů pro všechny zůčastněné
  a to bez výjimky (+bonus v podobě návštěvy u disciplinární komise).
```

Poslední modifikace: 19. February 2024
Pokud naleznete na této stránce chybu, oznamte to dopisem na adresu peringer AT fit.vutbr.cz