x86 Assembler Übung: Liste von Primzahlen

Klaus Kusche

Gesucht ist eine Assembler-Funktion, die *alle Primzahlen* bis zu einer als Parameter übergebenen Grenze mit **`printf`** *ausgibt* und deren *Anzahl als Returnwert* zurückliefert (es gibt dazu wieder ein vorgegebenes Hauptprogramm).

Dazu soll die Funktion intern den Algorithmus ***„Sieb des Erathostenes“*** verwenden, und zwar auf einem mit **`calloc`** in der richtigen Größe *dynamisch angelegten* ***Bit-Array***: Eine **`0`** im **`n`**-ten Bit bedeutet *„**`n`** ist prim“*, eine **`1`** bedeutet *„nicht prim“*:

In C würde die Funktion in etwa wie folgt aussehen:

```
unsigned int primes(unsigned int limit)
{
    int anzahl = 0;
    // Für die calloc-Größe: Bit-Anzahl auf Bytes umrechnen und aufrunden!
    void *bits = calloc(limit / 8 + 1, 1);
    if (bits != NULL) {
        for (unsigned int n = 2; n <= limit; ++n) {
            if („n-tes Bit in bits gleich 0“) {
                printf("%u ", n);
                ++anzahl;
                for (unsigned int i = n + n; i <= limit; i += n) {
                    „Setze i-tes Bit in bits auf 1“;
                }
            }
        }
        free(bits);
    }
    return anzahl;
}
```

Hinweise:

- Rechne intern mit *64 bit* **`int`**'s, damit es kein Problem mit Überläufen gibt!
- Erinnere dich, wie man eine Funktion *aufruft*:
  Wert des *ersten Parameters* ins **`rcx`** und des *zweiten Parameters* ins **`rdx`** legen,
  Funktion mit **`call`**-Befehl aufrufen, *Returnwert* liegt danach im **`rax`**.
- **Bit-Operationen:**

  Die x86-Architektur hat *eigene Befehle für Bit-Arrays*:
  Wir brauchen **`bt`** zum *Testen eines Bits* (das kopiert das Bit ins *Carry-Flag*,
  man prüft also danach mit **`jc`** oder **`jnc`** den Carry), und **`bts`** zum *Setzen eines Bits*.

  Alle Bit-Array-Befehle haben *zwei Operanden*:

  - Der *erste Operand* ist in unserem Fall ein *Speicher-Operand* (also in **`[ ]`** !),
    nämlich die *Adresse*, an der das Bit-Array beginnt.
  - Der *zweite Operand* ist ein *Register*, nämlich die *Nummer des Bits* im Bit-Array.

- **Stack:**

  Wir rufen diesmal C-Funktionen auf. Das hat 2 wichtige Konsequenzen:

  - Wir brauchen einen ***„vorschriftsmäßigen" Stack***:
    Stack-Pointer mit *16-Byte-Alignment*, 4*8 Bytes *Parameter-Area*, …
    Dazu ist es hilfreich, auch gleich ein Register als *Frame-Pointer* zu verwenden.
  - Die aufgerufenen Funktionen *überschreiben* die „frei verwendbaren" Register.
    Wir müssen daher alle *unsere Werte*, die die Aufrufe „überleben" sollen
    (alles außer dem **`i`** oben), in ***„Callee saves"-Registern*** unterbringen.
    Dafür müssen wir die Aufrufer-Werte in den benutzten „Callee saves"-Register
    gleich beim Aufruf unserer Funktion *sichern* und am Ende *wiederherstellen*.

  Im Detail:

  - Ich habe folgende Register benutzt:
    - **`rbp`**: ***Frame Pointer***
    - **`r12`**: **`bits`**
    - **`r13`**: **`n`**
    - **`r14`**: **`limit`** (weil wir den Parameter ja aus **`rcx`** wegsichern müssen!)
    - **`r15`**: **`anzahl`** (wohin kopierst du das am Ende der Funktion?)

    Diese fünf Register werden *gleich am Anfang* der Funktion mit **`push`**-Befehlen
    auf den Stack gesichert und *am Ende* unmittelbar vor dem **`ret`**
    in umgekehrter Reihenfolge mit **`pop`**-Befehlen *wiederhergestellt*.
  - Unmittelbar *nach* den fünf **`push`** wird der aktuelle **`rsp`** als *Frame-Pointer*
    gespeichert: **`mov rbp, rsp`**

    Und unmittelbar *vor* den fünf **`pop`** wird der **`rsp`** wiederhergestellt:
    `mov rsp, rbp`
  - Dann wird der **`rsp`** auf eine *durch 16 teilbare Adresse abgerundet*:
    `and rsp, ~0xf` (*Warum* hat das den gewünschten Effekt?)
  - Danach werden *32 Bytes Parameter-Area* für die aufgerufenen Funktionen
    reserviert: **`sub rsp, 32`**

  Erst nach diesem „Vorspiel" beginnt der eigentliche Code
  mit der Initialisierung unserer Register und dem **`calloc`**-Aufruf.

  Anmerkung: Wir müssten noch deutlich mehr tun, um unsere Funktion
  Debugger-fähig und *Exception-fähig* zu machen. Darauf *verzichten* wir aber.

- Wir verwenden **`calloc`**, nicht **`malloc`**, weil **`calloc`** den allokierten Speicher
  gleich *auf 0 initialisiert*, was uns Arbeit spart.

  `calloc` ist für Arrays gedacht und erwartet daher *zwei Parameter*:
  Die *Anzahl der Array-Elemente* und die *Größe eines Elementes* (in *Bytes*),
  wobei es für uns egal ist, ob wir ein Element mit vielen Bytes
  oder viele Elemente mit je einem Byte anlegen.

  Überlege, wie du das **`/8`** für die Umrechnung von der Anzahl von Bits
  auf die Anzahl von Bytes *ohne Divisionsbefehl* erledigst!

  `calloc` liefert wie **`malloc`** einen *Pointer auf den allokierten Block* als Returnwert,
  der auf **`NULL`** (= *„kein Platz"*) geprüft und am Ende des Programms
  wieder mit **`free`** freigegeben werden sollte.

- **`printf`**-Formatstring:
  - Zum Anlegen von *C-String-Werten* im Speicher gibt es den Assembler-Befehl **`.asciz`**, er hängt automatisch ein **`\0`** an.
  - Strings gehören so wie globale Variablen in das *Segment* **`.data`**, nicht **`.text`**.
  - Wenn **`format`** das Label deines Strings ist, sollte laut Hausverstand ein **`mov rcx, format`** vor dem **`printf`**-Call funktionieren, um die Adresse des Strings als Parameter zu laden. Das klappt aber *nicht*, weil der Linker die im **`mov`** versteckte Adresse nicht richtig anpasst.

    Richtig macht man es mit **`lea rcx, [format]`** .
    `lea` steht für *„load effective adress“ („Lade die Adresse des Speicher-Operanden“*).
- **Tipp:** Wenn du die *Schleifenzähler* bei beiden Schleifen *vor der Schleife* um einen Schritt *niedriger initialisierst* und dafür *gleich am Anfang der Schleife* (vor dem Vergleich) statt am Schleifenende *hochzählst*, wird der Code um ein paar Befehle kürzer und die Sprung-Struktur etwas übersichtlicher!