Einführung in die Threadprogrammierung

Florian Mayer

Februar 2014

Zusammenfassung

Dieses Dokument ist als eine Einführung in die Welt der Threadprogrammierung zu verstehen. Dabei wird in den Beispielen konsequent C als Programmiersprache verwendet und davon ausgegangen, dass der Leser ein wesentliches Verständnis für die Funktionsweise von unixoiden Betriebssystemen besitzt.

Inhaltsverzeichnis

1 Allgemeines **1**
- 1.1 Kernel- und User-Threads . . . 1
- 1.2 Implementierungen . . . 2
- 1.3 Plattformunabhängigkeit . . . 2

2 Einführung in POSIX-Threads **2**
- 2.1 Thread-Management . . . 2
  - 2.1.1 Funktionen . . . 2
  - 2.1.2 Sourcecodebeispiel . . . 3
- 2.2 Wechselseitiger Ausschluss (Mutual Exclusion) . . . 3
  - 2.2.1 Funktionen . . . 3
  - 2.2.2 Sourcecodebeispiel . . . 4
- 2.3 Bedingungsvariablen . . . 5
  - 2.3.1 Funktionen für Bedingungsvariablen . . . 5

3 Nachbildung von zählenden Semaphoren **6**
- 3.1 Schematische Nachbildung . . . 6
- 3.2 Implementierung mithilfe von Pthreads . . . 7

4 Verwendung von nativen zählenden Semaphoren (POSIX) **9**
- 4.1 Funktionen . . . 9
- 4.2 Compilerflags . . . 9
- 4.3 Sourcecodebeispiel . . . 10

1 Allgemeines

1.1 Kernel- und User-Threads

Es wird zwischen sog. User- und Kernel-Threads unterschieden. Der POSIX Standard Pthreads definiert einen Satz von C-Objekten und -Funktionen, die plattformunabhängiges Programmieren mit Kernel-Unterstützung bieten. Kernel-Threads werden vom Betriebssystem verwaltet. Hierbei besitzt ein Prozess entweder einen oder mehrere Threads, die jeweils innerhalb eines Schedulings ablaufen. Jeder Prozess hat nach wie vor seinen eigenen Speicherbereich, geöffnete Dateien, ein Daten- und ein Codesegment. Wenn ein Prozess nur einen Kernel-Thread besitzt, verfügt er nur über einen Befehlszähler und ein Stacksegment. Wenn hingegen mehrere Kernel-Threads innerhalb des Prozesses

ausgeführt werden, wird für jeden Thread ein separater Befehlszähler bzw. ein separates Stacksegment mitgeführt. Einen Prozess mit n Threads nennt man Task.
User-Threads oder Userland-Threads sind keine Threads im Sinne der obigen Darstellung. Sie ermöglichen zwar die "parallele" Abarbeitung von Anweisungen, die Verwaltungsalgorithmen hierfür sind jedoch nicht direkt im Kernel implementiert. Der Kontextwechsel zwischen den einzelnen Userland-Threads übernimmt nun eine Thread-Bibliothek außerhalb des Kernellands. Diese verfügt offensichtlich auch über einen eigenen Scheduler. Aus diesem Grund weiß das Betriebssystem nichts von der Existenz aktiver Userland-Threads. Daher, und weil die Userland-Bibliothek nicht im priviligierten Prozessormodus ausgeführt wird, kann nur kooperatives Scheduling betrieben werden. Kernel-Threads werden hingegen präemptiv im Kernel Verwaltet.

1.2 Implementierungen

- Userland-Threads
  - Linux: LinuxThreads (nicht mehr weiterentwickelt)
  - Windows: seit Windows 98 Fibers
  - JVM verwendet Green Threads, falls der JVM-Host keine Kernel-Threads bietet. Ansonsten greift die JAVA-Runtime auf Betriebssystemeigene Threading-Bibliotheken zurück. Dieses Verhalten kann ggf. vor dem Übersetzen konfiguriert werden.
- Kernel-Threads
  - Linux: Native Posix Thread Library (NPTL). Pthreads schalt NPTL ein und versteckt plattformspezifische Details..
  - Windows: Funktionen der win32/64-API

1.3 Plattformunabhängigkeit

Nahezu alle modernen UNIX basierten Betriebssysteme halten sich grundlegend an den POSIX-Standard. [1] Die Impelemntierungen der Pthreads-Library unterscheiden sich stark im Hinblick auf Standardwerte. Beispielsweise könnte die Größe des Stacks für Thread-Routinen auf Plattform A 2^{13} Byte, auf Plattform B jedoch lediglich 2^{10} Byte betragen. Der vollständige POSIX-Standard kann auf der Webseite opengroups.org eingesehen werden. [2]

2 Einführung in POSIX-Threads

Im Folgenden werden zunächst die meistgenutzten Funktionen in vier Kategorien näher beschrieben. Die Bezeichner aller Funktionen haben das Präfix "pthread_". Die exakte Syntax jeder Pthreads-Funktion kann mithilfe des Kommandos "man" unter UNIX/Linux-Systemen abgefragt werden. Beispiel: "man pthread_create" Außerdem sei eine sehr ausführliche Referenz empfohlen. [3] Die folgenden kurzbeschreibungen beschränken sich auf die nötigsten Informationen. Insbesondere werden die meisten Fehlerrückgabewerte nicht beschrieben. Hierzu sollte die Online-Referenz konsultiert werden.

2.1 Thread-Management

2.1.1 Funktionen

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attribute, void *(*function)(void *), void *args)`

[1] Ausnahmen bestätigen auch hier wie immer die Regel. Viele UNIX-artige Kernel orientieren sich an der POSIX-Spezifikation, jedoch ist z.B. Linux POSIX "nur weitgehend kompatibel" Im Falle von Linux wurden jedoch viele POSIX vorschriften ignoriert, da sie historisch bedingt limitierend wirken.

[2] http://www.opengroup.org/onlinepubs/007904975/toc.htm

[3] http://cursuri.cs.pub.ro/ apc/2003/resources/pthreads/uguide/document.htm

Diese Routine dient der Erstellung von Threads. Die Funktion gibt 0 zurück, wenn die Ausführung erfolgreich war, ansonsten einen Fehlercode ungleich 0. Returnstatements in einer in einem Thread ausgeführten Funktion beenden den jeweiligen Thread. Der Aufruf von `void exit(x)` innerhalb eines Threads beendet den gesamten Prozess.

- `void pthread_exit(void *return)`
  Ein derartiger Aufruf führte zur Beendigung des aufrufenden Threads.

- `void pthread_join(pthread_t *thread, void **thread_return)`
  Eine Routine zum Warten auf die Beendigung des gestarteten Threads `pthread_t * thread`.

- `pthread_t pthread_self()`
  Pthread_self ist die Pthreads-Analogie zu `int getid()`

- `int pthread_cancel(pthread_t thread)`
  Diese Prozedur fordert das System dazu auf den angegebenen Thread zu beenden. Die Exit-Handler des entsprechenden Threads (sofern eingerichtet) werden ausgeführt und die Funktion kehrt unmittelbar nach Absetzen der Terminierungsanforderung zurück, da die Aufräumarbeiten asynchron im Kontext des zu beendenden Threads ausgeführt werden. Es gibt nur einen Fehlercode, den die Funktion im Fehlerfall zurückgegeben wird (ERSCH, vgl. man pthread_cancel). Ansonsten wird, wie gewöhnlich, mit dem Wert 0 die erfolgreiche Ausführung signalisiert.

2.1.2 Sourcecodebeispiel

```
1  #include <pthread.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <string.h>
5
6  void *printNum(void *num){
7      for(;;);
8      return NULL;
9  }
10
11 int main(int argc, char ** argv){
12     int i;
13     pthread_t thread[8];
14
15     /* creation of eight threads */
16     for(i=0; i<8; i++){
17         /* first NULL means that we want to use the standard attributes for
               our threads
18             second NULL means that we do not need parameters */
19         pthread_create(thread+i, NULL, printNum, (void *) NULL);
20     }
21
22     /* Threads won't ever return */
23     for(i=0; i<8; i++){
24         pthread_join(thread[i], NULL);
25     }
26
27     return EXIT_SUCCESS;
28 }
```

2.2 Wechselseitiger Ausschluss (Mutual Exclusion)

2.2.1 Funktionen

Die Pthreads-Bibliothek stellt einen Mechanisumus für Mutexe bereit. Die entsprechenden Funktionen sind mit pthreads_mutex geprefixt. Die wichtigesten Funktionen sind:

- `int pthread_mutex_init(pthreads_mutex_t *mutex, pthread_mutexattr_t *attr)`
  Initialisert einen Mutex. `pthread_mutexattr_t *attr` kann dabei zur Konfiguration verwendet werden. Das Setzen von attr = NULL genügt, wenn die Standardwerte ausreichend sind. Als Parameter `pthread_mutext_t *mutex` muss ein bereits initialisierter Zeiger mit der richtigen Größe übergeben werden, denn die Funktion selbst reserviert keinen Speicher!

- `int pthread_mutex_destroy(pthread_mutex_t *mutex)`
  Entfernt einen erstellten Mutex aus dem System. Die Funktion gibt 0 zurück, wenn der Aufruf erfolgreich war, ansonsten wird ein Fehlercode zurückgegeben. Mögliche Werte hierfür sind z.B.:
  - EBUSY: Mutex ist noch nicht wieder freigegeben worden.
  - EINVAL: Es wurde eine Adresse übergeben, die keinen gültigen Mutex referenziert.

- `int pthread_mutex_trylock(pthread_mutex_t *m)`
  Versucht den Mutex zu schließen. Die Routine gibt 0 zurück, wenn der Mutex erfolgreich geschlossen werden konnte, ansonsten wird ein Fehlercode zurückgegeben. Wichtige Werte hierfür sind wiederum:
  - EINVAL: Der Pointer auf den Mutex ist ungültig.
  - EBUSY: Mutex konnte nicht geschlossen werden, da er zum Zeitpunkt des Funktionsaufrufs schon geschlossen war.

- `int pthread_mutex_timedlock(pthread_mutex_t *restrict m, const struct timespec *restrict timeout)`
  Verhält sich wie `pthread_mutex_lock()`, mit dem Unterschied, dass nur `timeout` Zeitheinheiten gewartet wird. Wenn es nicht möglich war den Mutex zu schließen, wird ETIMEDOUT zurückgegeben.

- `int pthread_mutex_lock(pthread_utext_t *restrict m)`
  Schließt einen Mutex. Wenn dieser bereits geschlossen ist, blockiert die Funktion solange, bis der Thread, der den Mutex hält, diesen wieder freigibt. Auch hier wird 0 bei Erfolg und ein Fehlercode bei einem Fehler zurückgegeben. Werte für den Fehlercode sind z.B.:
  EINVAL: Falsche Parameter

- `int pthread_mutex_unlock(pthread_mutex_t *m)`
  Gibt einen geschlossenen Mutex wieder frei.

- `int pthread_mutexattr_init(pthread_mutex_t *m)`
  Initialisiert ein Attributobjekt vom Typ `pthread_mutexattr_t` . [4]

- `int pthread_mutexattr_destroy(pthread_mutex_t *m)`
  Zerstört ein Mutexattributobjekt.

2.2.2 Sourcecodebeispiel

```
1  /* Creates one(!) file pointer every thread has access to.
2      Synchronisation happens with the mutex sync which is initialized
3      dynamically in the main function. */
4  #include <pthread.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7
8  #define FPATH "resource"
9
10 /* parameters */
11 struct f {
```

[4] vgl. man pthread_mutexattr_init

```
12      FILE *fptr;
13      pthread_mutex_t *mut;
14      int tid;
15  };
16
17  /* routine executed by threads */
18  void *writeSth(void *data){
19      struct f temp = *((struct f *) data);
20      pthread_mutex_lock(temp.mut);
21      fprintf(temp.fptr, "I␣am␣thread␣with␣number:␣%d\n", temp.tid);
22      pthread_mutex_unlock(temp.mut);
23      return NULL;
24  }
25
26  int main(int argc, char ** argv){
27      int i;
28      struct f data[8];
29      FILE *file = fopen(FPATH, "w+");
30      pthread_t thread[8];
31
32      /* create the mutex */
33      pthread_mutex_t sync;
34      pthread_mutex_init(&sync, NULL);
35
36      /* starts threads and passes arguments */
37      for(i=0; i<8; i++){
38          data[i].fptr = file;
39          data[i].tid = i;
40          data[i].mut = &sync;
41
42          pthread_create(thread+i, NULL, writeSth, (void *) (data+i));
43      }
44
45      /* waits for all threads */
46      for(i=0; i<8; i++){
47          pthread_join(thread[i], NULL);
48      }
49
50      /* free the resources taken by the mutex */
51      pthread_mutex_destroy(&sync);
52      fclose(file);
53      return EXIT_SUCCESS;
54  }
```

2.3 Bedingungsvariablen

2.3.1 Funktionen für Bedingungsvariablen

Pthread-Mutexe verhalten sich wie Semaphore, die mit dem Wert 1 initialisiert wurden. Nur ein einziger Thread kann bei dieser Art der Synchronsiation zu einer beliebigen Zeit t den kritischen Abschnitt betreten. Bei komplexeren Problemen, wie z.B. dem Erzeuger-Verbraucher-Problem, ist es jedoch notwendig, bis zu n Threads gleichzeitig in einem zu synchronisierenden Abschnitt arbeiten zu lassen. Dies wird durch die Kombination von Mutexen und Bedingungsvariablen ermöglicht. Um Bedingungsvariablen zu verwenden sind folgende Schritte notwendig:

1. Vereinbarung des zu synchronisierenden globalen Speichers.
2. Initialisierung einer Bedingungsvariable (dargestellt als Objekt der Pthreads-Bibliothek).
3. Initialisierung eines Mutexes mit Bezug auf die Bedingungsvariablenerstellung der Threads.

Pthreads bietet folgende Funktionen für die Erstellung, Bearbeitung und Vernichtung von Bedingungsvariablen:

- `int pthreads_cond_init(pthread_cond_t *c, const pthread_condattr_t *restrict a)`
  Initialisert ein zuvor deklariertes Objekt vom Typ `pthread_cond_t`. Dabei kann ein Attribut-Objekt übergeben werden, welches weitere Feineinstellungen innerhalb der Pthread-Bibliothek ermöglicht. Die Initialisierung dieser Objekte ist weiter unten beschrieben.

- `int pthread_cond_destroy(pthread_cond_t *c)`
  Gibt die von `pthread_cond_t *c` benötigten Ressourcen wieder frei.

- `int pthread_condattr_init(pthread_condattr_t *c)`
  Initialisert ein zuvor deklariertes Objekt vom Typ `pthread_condattr_t`. Für Bedingungsvariablen ist nur das Attribut "process-shared" definiert. Wenn eine Bedingungsvariable mit diesem Attribut initialisiert wird, ist sie Prozessübergreifend sichtbar.

- `int pthread_condattr_destroy(pthread_condattr_t *c)`
  Entfernt ein Bedingungsvariablenattribut.

- `int pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)`
  Sollte nur aufgerufen werden, wenn Mutex `pthread_mutext_t *m` gesperrt ist. Beim Aufruf wird der Mutex automatisch frei und der aktuelle Thread blockiert. Die Blockade hält solange an, bis das Signal [5] empfangen wird. Der Mutex `pthread_mutex_t` wird, nachdem die Funktion fertigt ausgeführt wurde, automatisch gesperrt.

- `int pthread_cond_signal(pthread_cond_t *c)`
  Wird verwendet, um einen aktuell mittels `pthread_cond_wait(c,m)` wartenden Thread aufzuwecken. Die Funktion sollte nur aufgerufen werden, wenn der Mutex gesperrt ist. Des Weiteren muss nach dem Aufruf zusätzlich der Mutex entsperrt werden.

- `int pthread_cond_broadcast(pthread_cont_t *c)`
  Wird wie die vorherige Funktion verwendet, mit der Ausnahme, dass nun beliebig viele wartende Threads angesprochen werden können.

3 Nachbildung von zählenden Semaphoren

Mithilfe von Mutexen, Barrieren und Bedingungsvariablen können in der Threadprogrammierung die meisten Synchronisationsprobleme gelöst werden. Es gibt jedoch einige Fälle in denen man gerne auf "klassische" zählende Semaphore zurückgreifen möchte oder muss. Wenn man z.B. eine Altsoftware optimieren und an aktuelle Gegebenheiten anpassen möchte, bieten sich Threads aufgrund ihrer leichtgewichtigkeit besonders an. Damit nun keine vollständige Restrukturierung nötig wird, passt man nur die Implementierung der Semaphoroperationen (p, v, seminit, semdest) an und verwendet weiterhin diese Funktionen als Synchronisationsmittel innerhalb des Programms.

3.1 Schematische Nachbildung

Im Folgenden wird Pseudo-Code gezeigt, der das grundlegende Verfahren bei der Nachbildung von zählenden Semaphoren erläutern soll.

```
1 /* Struktur fuer ein zaehlendes Semaphor */
2 struct sem_zs {
3     // Zur Realisierung eines Schutzes von "zaehler" und "warten".
4     struct semaphor mutex;
5     // Binaeres Semaphor zur Realisierung des Wartens und Weckens
6     struct semaphor sem;
7
```

[5] Pro Instanz einer Bedingungsvariable gibt es eine Signalquelle, die mit ihr assoziiert ist

```
8       // Bekannter Semaphor-Zaehler
9       int zaehler;
10      // Anzahl Wartender
11      int warten;
12  };
13
14  /* Mit "n" als Initialisierungswert fuer zaehlendes Semaphor */
15  seminit_z(struct sem_zs s, int n) {
16      seminit(s.sem, 0);
17      seminit(s.mutex, 1);
18      s.warten = 0;
19      s.zaehler = n;
20  }
21
22  /* P-Operation fuer zaehlendes Semaphor */
23  pop_z(struct sem_zs s){
24      P(s.mutex);
25      if (s.zaehler > 0) {
26          s.zaehler--;
27          V( s.mutex);
28      } else {
29          s.warten++;      // Anzahl Wartender erhoeht sich
30          V(s.mutex);
31          P(s.sem);        // warten
32      }
33  }
34
35  /* V-Operation fuer zaehlendes Semaphor */
36  vop_z(struct sem_zs s){
37      P(s.mutex);
38      if(s.warten > 0) {
39          s.warten--;
40          V(s.sem);   // aufwecken
41      } else {
42              s.zaehler++;
43      }
44
45      V(s.mutex);
46  }
```

3.2 Implementierung mithilfe von Pthreads

```
1   /* Bildet zaehlende Semaphore nach */
2   struct sem_zs {
3       pthread_mutex_t mutex;
4       pthread_mutex_t sem;
5       int cnt;
6       int wait;
7   };
8
9
10  /***********************/
11  /* Funktionsprototypen */
12  /***********************/
13  /* P/V-Operation auf ein zaehlendes Sem.
14      Param: sem = Initialisiertes (!) semaphor
15      Return: 0 bei Erfolg und 1 bei Fehler*/
16  int v(struct sem_zs *sem);
17  int p(struct sem_zs *sem);
18
```

```
19 /* Erzeugt und initialisiert ein Semaphor.
20     Param: num = Specifies the initial value for the Semaphor
21     Return: NULL bei Fehler und jeder andere Wert bei Erfolg */
22 struct sem_zs *initsem(int num);
23
24 /* Zerstoert ein Semaphor
25     Param: sem = Specifies the semaphor one wants to destroy
26     Return: 0 bei Erfolg and 1 bei Feiler */
27 int destsem(struct sem_zs *sem);
28
29 /***********************/
30 /* Implementierung     */
31 /***********************/
32
33 struct sem_zs *initsem(int num) {
34     if(num < 0)
35         return NULL;
36
37     struct sem_zs *res = malloc(sizeof(struct sem_zs));
38     if(res == NULL){
39         return NULL;
40     }
41
42     res->wait = 0;
43     res->cnt = num;
44
45     pthread_mutex_init(&(res->sem), NULL);
46     pthread_mutex_init(&(res->mutex), NULL);
47
48     pthread_mutex_lock(&(res->sem));
49
50     return res;
51 }
52
53 int destsem(struct sem_zs *sem) {
54     if(sem == NULL)
55         return 1;
56
57     pthread_mutex_destroy(&(sem->sem));
58     pthread_mutex_destroy(&(sem->mutex));
59
60     free(sem);
61
62     return 0;
63 }
64
65 int p(struct sem_zs *sem) {
66     if (sem == NULL)
67         return 1;
68
69     pthread_mutex_lock(&(sem->mutex));
70
71     if (sem->cnt > 0){
72         sem->cnt--;
73         pthread_mutex_unlock(&(sem->mutex));
74     } else {
75         sem->wait++;
76         pthread_mutex_unlock(&(sem->mutex));
77         pthread_mutex_lock(&(sem->sem));
78     }
79
```

```
80      return 0;
81  }
82
83  int v(struct sem_zs *sem) {
84      if (sem == NULL)
85          return 1;
86
87      pthread_mutex_lock(&(sem->mutex));
88
89      if(sem->wait > 0){
90          sem->wait--;
91          pthread_mutex_unlock(&(sem->sem));
92      } else {
93          sem->cnt++;
94      }
95
96      pthread_mutex_unlock(&(sem->mutex));
97
98      return 0;
99  }
```

4 Verwendung von nativen zählenden Semaphoren (POSIX)

Im vorangegangenen Kapitel wurde gezeigt, wie mit einfachen Mitteln aus Pthread-Mutexen und den zugehörigen Operationen, zählende Semaphore inclusive der bekannten P- und V-Operationen nachgebildet werden können. Dies ist jedoch nur sinnvoll, wenn man auf einem System arbeitet, das keine vernünftige, aus Threads heraus verwendbare, Semaphor-Implementierung bietet. Zum Glück entsprechen heute die meisten UNIX-artigen Betriebssysteme der POSIX-Spezifikation. Auf kompatiblen Systemen findet man die Header-Datei “semaphore.h”, welche die API für POSIX-Semaphore definiert.

4.1 Funktionen

- `int sem_open(sem_t *sem, int pshared, unsigned int value)`
  Initialisiert ein Semaphor mit dem Wert `unsigned int value`. Wenn `int pshared` auf den Wert 1 gesetzt wird, kann das Semaphor von verschiedenen Prozessen genutzt werden. Ansonsten steht es lediglich den Threads innerhalb eines Prozesses zur Verfügung. Der Benutzer der Funktion ist selbst dafür verantwortlich, dass die Struktur `sem_t *sem` an einer geeigneten Stelle im Speicher platziert wird. Insbesondere ist darauf zu achten, dass auf die Struktur bei `pshared = 1` von mehreren Prozessen zugegriffen werden kann (vlg. shm-Segmente).

- `int sem_wait(sem_t *sem);`
  Blockiert den Aufrufer solange, bis das Semaphore von anderer Stelle mihtilfe der Funktion sem_post (siehe unten) freigegeben wurde.

- `int sem_trywait(sem_t *sem);`
  Wie `sem_wait`, mit folgendem Unterschied: `sem_trywait()` kehrt sofort mit einem Fehlercode zurück, wenn die Dekrementierungsoperation nicht durchgeführt werden konnte, wenn also `sem` den Wert 0 hatte. Der Fehlercode ist: EAGAIN

- `int sem_post(sem_t *sem);`
  Inkrementiert das Semaphor `sem_t *sem` oder kehrt mit einem Fehlercode zurück. Eine häufige Ursache hierfür ist z.B. `sem == NULL`;

4.2 Compilerflags

Es muss mit “-lrt” gebaut werden.

4.3 Sourcecodebeispiel

```c
#include <semaphore.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define THREADS 20

sem_t ok;
int milk;

void *buyer(void *arg){
    /* P-Op */
    sem_wait(&ok);

    milk++;

    sem_post(&ok);

    return NULL;
}

int main(int argc, char **argv){
    pthread_t threads[THREADS];
    int i;
    milk = 0;

    int(sem_init(&ok, 0, 1)){
        fprintf(stderr, "[main()] Could not initialize the semaphore!\n");
        return EXIT_FAILURE;
    }

    for(i=0; i<THREADS; i++){
        if(pthreads_create(&threads[i], NULL, buyer, NULL)){
            fprintf(stderr, "[main()]" Could not start threads!\n);
            return EXIT_FAILURE;
        }
    }

    for(i=0; i<THREADS; i++){
        if(pthreads_join(threads[i], NULL)){
            fprintf(stderr, "[main()] Could not join the thread %d!\n");
            return EXIT_FAILURE;
        }
    }

    sem_destroy(&ok);
    printf("Total amount of milk units: %d\n", milk);

    return EXIT_SUCCESS;
}
```