

# **C Grundlagen - Threads**

Michael Strassberger  
saremox@linux.com

3. Juli 2014

# Inhaltsverzeichnis

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Problems that can be seperated . . . . .	3
1.2	problems that canno't be seperated . . . . .	3
<b>2</b>	<b>What are threads</b>	<b>3</b>
2.1	With great power comes great responsibilitly . . . . .	4
<b>3</b>	<b>How to use threads in C</b>	<b>5</b>
3.1	pthread . . . . .	5
3.1.1	Compiling . . . . .	5
3.1.2	Creating threads . . . . .	6
3.1.3	passing Arguments & Joining . . . . .	7
3.1.4	Pthread Mutexes . . . . .	8
3.2	openmp . . . . .	10
3.2.1	Hello World . . . . .	10
<b>4</b>	<b>Literature</b>	<b>10</b>

# 1 Introduction

We are hitting physical limits of computing speed on single core machines. Since we're hitting these limits hardware manufacturer began to invent additional instruction sets to optimize CPU cycles like the MMX IS. But we're also facing a new trend to Multicore systems even on mobile devices. To take advantage of these developments we should first take a look at what general computing problems we face.

## 1.1 Problems that can be separated

These are generally problems we can split into smaller autonomous tasks. Like simulating the weather of a specific chunk of the world. We can split this chunk into smaller ones and compute each chunk on a processing core. But we have also make sure that every processor gets all results to compute the next iteration. Every problem that can be processed like this can benefit from multiple core machines.

## 1.2 problems that can't be separated

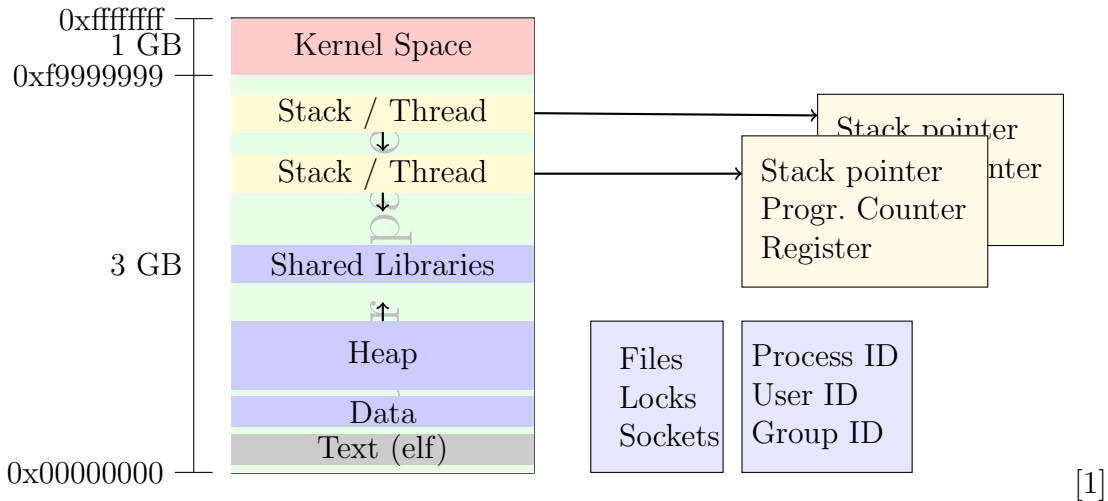
These are usually tasks that don't need much processing power or depend on previous results to compute. But also hardware access is limited to be done with only one processor ( In some cases it's possible, but we'll assume that it's not for easier understanding ). But it's not unlikely to split these tasks across processing units. Let's take a look at a 4 core processing unit and an example scenario

1. Hardware Access hard drive - saving / reading file
2. Sorting and preparing data from 1.
3. Do computing with the data
4. Showing graphical user interface

# 2 What are threads

*In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler. [2]*

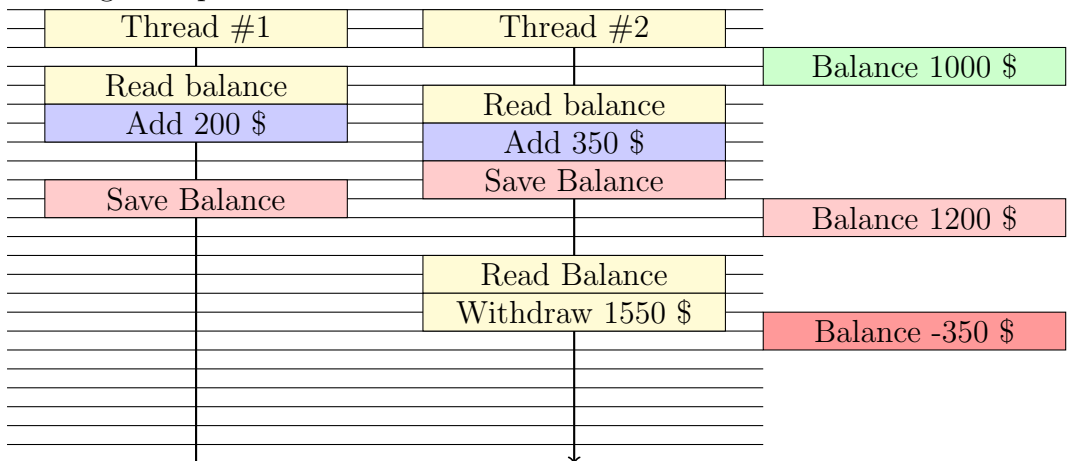
That implies that a thread only contains the minimum requirements to run independent from the main process. They also share their Memory so we can easily access the data of another thread.



In general a process consists of a Process ID and the UID GID with which it is running. It also has File descriptors, sockets and locks. The program code and static variables mapped to the bottom of our example memory model. In the middle the growing heap for dynamic memory allocation and the shared library code get loaded. At the very top of the memory we have after an random offset the Stack of our main thread. If we start another thread we'll only create a another stack. Starting a new thread requires only a minimum memory allocation that makes it not only very lightweight in memory, it also is very fast to create a thread. It's about 4-20 times faster than creating a new process. [1]

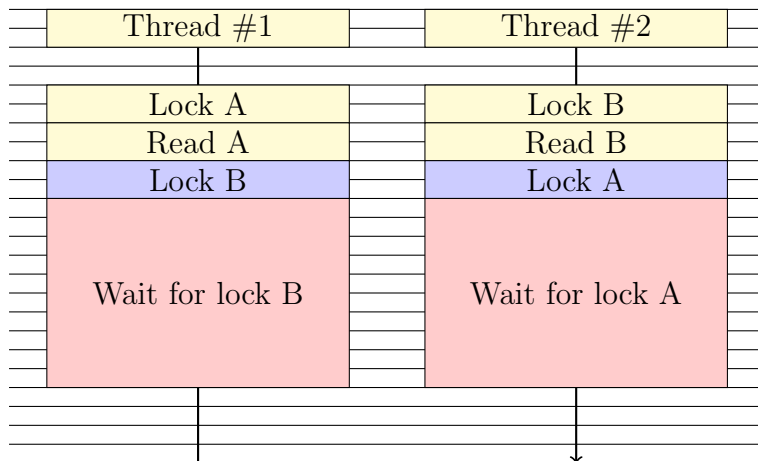
## 2.1 With great power comes great responsibility

Since we now know that threads share their heap memory, they can both access and write to it. This could lead to inconsistent data then we have to write operations at the same time on the same memory segment. This is a race condition. Take a look at a banking example.



We've lost 350 \$ because we try to write to the same memory at the same time. This can be prevented by using locking mechanism like Mutual exclusions. Mutexes can only

be obtained by one thread at a time, that makes it a very good solution for managing access to shared memory. But we've to ensure that we lock and unlock the mutexes. We have also to take into mind that this technique bring up some new problem: The deadlock.



We're now locking our resources that we use, but if someone else also wanting these Resources it could happen that both threads will wait for each other to release a specific resource they need to fulfill their task. This can be prevented by using techniques for looking if a lock is already obtained by someone else and using this for additional logic to prevent a deadlock.

### 3 How to use threads in C

8 There exist several threading libraries that can make your live more or less easy with threads. We've already heard about c11 threads and glib threads. We'll take a look at the Posix thread implementation and the easy to use openmp library.

#### 3.1 pthread

##### 3.1.1 Compiling

```
GCC
gcc -std=c11 -lpthread |file| -o |output|
```

```
MakeFile
1 Account : account.o
2 cc -lpthread -O3 account.o -o Account
3 account.o : main.c
4 cc -std=c11 -c -lpthread -lm -O3 main.c -o account.o
```

### 3.1.2 Creating threads

#### Quick overview

#### Pthread types

```
1 pthread_t thread
```

#### Pthread function calls

```
1 pthread_create (thread, attributes, start_routine, argument)
2 pthread_exit (status)
3 pthread_cancel (thread)
```

This is the minimum set of types and function calls to create threads and shut them down. To create a thread we have to pass a `pthread_t` typed variable that represents the thread and we also have to pass an function pointer (see presentation of Phillip Gawehn). Arguments and attributes are optional and you'll don't need them to create a thread. With `pthread_cancel` we can shut down a thread in our main thread, if the thread is hanging or doing wrong things. `pthread_exit` is used in a main thread to wait for all threads to terminate, otherwise then the main thread exits, all threads get killed

#### Hello world - Pthread Example

```
1 void * sayhello(void * arg)
2 {
3     long threadID = (long) arg;
4     fprintf(stdout, "Hello from Thread #%d\n", threadID);
5     return 0;
6 }
7 int main(int argc, char* argv[])
8 {
9     pthread_t thread[20];
10    for(long threadid = 1; threadid < 20; threadid++)
11        pthread_create(&thread[threadid], NULL,
12                    sayhello, (void *)threadid);
13    pthread_exit(NULL);
14 }
```

We'll create in this example 20 Threads and each of them will print on screen "Hello from Thread #xx" in nearly random order. In this example we can see that we can't determine then a thread gets executed.

#### Ausgabe

```
1 Hello from Thread #6
2 Hello from Thread #7
3 Hello from Thread #9
4 Hello from Thread #10
5 Hello from Thread #8
```

### 3.1.3 passing Arguments & Joining

#### Job Struct & Worker Function

```
1 struct job {
2     int start;
3     int end;
4     unsigned long long int result;
5 };
6
7 void * function(void *ptr) {
8     struct job *myJob = (struct job*) ptr;
9     for(int i = myJob->start ; i <= myJob->end ; i++)
10    {
11        myJob->result += (5*(i*i) + 5);
12    }
13 }
```

This is a typical setup in programming with threads. We've a data structure that represents all data needed for the thread to do a specific job. And a function that executes this job.

#### Main Function

```
1 int main() {
2     struct job job1 = {0,500,0};
3     struct job job2 = {501,1000,0};
4     pthread_t worker1;
5     pthread_t worker2;
6
7     pthread_create(&worker1,NULL,function,(void *) &job1);
8     pthread_create(&worker2,NULL,function,(void *) &job2);
9     pthread_join(worker1,NULL);
10    pthread_join(worker2,NULL);
11    fprintf(stdout,"Result: %llu + %llu = %llu \n",job1.result,job2.result,(job1.
12        result+job2.result));
13 }
```

We declare and initialize 2 jobs and declare 2 threads. Then we create the 2 worker threads to process our 2 jobs data structures. With pthread\_join we'll wait until the given thread has finished excuting. Now we can be shure that, no worker is working on the results and we can sum up the 2 sums to get our final result.

### 3.1.4 Pthread Mutexes

#### Quick Overview

##### Data types

```
1 pthread_mutex_t mutex
```

##### Creating Mutexes

```
1 pthread_mutex_init (mutex, attr)
2 pthread_mutex_destroy (mutex)
```

##### Locking and unlocking

```
1 pthread_mutex_lock (mutex)
2 pthread_mutex_trylock (mutex)
3 pthread_mutex_unlock (mutex)
```

With `pthread_mutex_init` we initialize a mutex, we can set attributes for this mutex, but we'll use the default behavior of our mutex therefore we can also use `PTHREAD_MUTEX_INITIALIZER` constant to get a Mutex. The locking and unlocking functions just locks and unlocks a mutex. With `trylock` we have a non blocking function call. `Trylock` returns instantly and if it's return value is zero we locked the mutex otherwise the mutex was already lock by another Thread. This is useful for preventing deadlocks.

#### Bank transfer

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <unistd.h>
4 #include <time.h>
5 #include <stdlib.h>
6 struct account{
7     char* name;
8     int accountNumber;
9     int balance;
10    pthread_mutex_t lock;
11 };
12
13 struct transferJob{
14     struct account *from;
15     struct account *to;
16     int ammount;
17 };
```

We've a bank account and transferjob data structure. We'll now simulate with them various transactions by using the mutex mechanics.



## Bank transfer

```
1 int transfer(struct account *from, struct account *to, int amount) {
2     pthread_mutex_lock(&from->lock);
3     if(from->balance > amount) {
4         if(pthread_mutex_trylock(&to->lock) == 0) {
5             from->balance -= amount;
6             to->balance += amount;
7             pthread_mutex_unlock(&to->lock);
8         } else {
9             pthread_mutex_unlock(&from->lock);
10            return transfer(from,to,amount);
11        }
12        pthread_mutex_unlock(&from->lock);
13        fprintf(stdout, "| %s\t | %d\t | > %d > | %s\t | %d\t |\n", from->name, from
14            ->balance, amount, to->name, to->balance);
15    } else {
16        pthread_mutex_unlock(&from->lock);
17        return -1;
18    }
19 }
```

This Function takes 2 banking accounts and transfers the given ammount from one to the other. First we acquire the Mutex of \*from and we check if the account is covered. If the account is enough funded we try to acquire the lock of \*to if we suceed we'll substracts ammont from \*from and add it to \*to. After this we'll free our \*to lock and free the \*from lock, printing a log message to the console and return. If we can't acquire the lock we free the \*from lock to prevent deadlocks and recursivly try to do the transaction again. If the account is not covered we'll free the \*from lock and return with -1 to inform the caller that we transaction was not executed.

## Bank transfer

```
1 int main(int argc, char **argv)
2 {
3     srand(time(NULL));
4
5     pthread_t thread1, thread2;
6     struct account accA = {"Max Mustermann", 1340005,
7         1000, PTHREAD_MUTEX_INITIALIZER};
8     struct account accB = {"Franz Mustermann", 1340005,
9         1000, PTHREAD_MUTEX_INITIALIZER};
10    struct transferJob job1 = {&accA, &accB, 20};
11    struct transferJob job2 = {&accB, &accA, 20};
12
13    pthread_create(&thread1, NULL, worker, &job1);
14    pthread_create(&thread2, NULL, worker, &job2);
15    pthread_join(thread1, NULL);
16    pthread_join(thread2, NULL);
17    fprintf(stdout, "%s account is at %d\n%s account is at %d\n", accA.name, accA.
18        balance, accB.name, accB.balance);
19    pthread_exit(NULL);
20 }
```

We initialize 2 bank accounts each with a mutex and we also create 2 jobs for our 2

worker thread model. We create these Thread and then wait with `pthread_join` until they finished. Then we print out the current account balances if they both are at 1000 nothing wrong had happened.

## 3.2 openmp

OpenMP is an easy to use API for using threads in your programs. You can use them in C / C++ and fortran applications. OpenMP is used by writing compiler directives where your application can be parralized.

### 3.2.1 Hello World

#### OpenMP Hello World

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     #pragma omp parallel
6     printf("Hello, world.\n");
7     return 0;
8 }
```

With `#pragma omp parrallel` we say openmp that we following can be parralized.

## 4 Afterword

You got an overview of that threads are and their concepts and how to use them in C with `pthread`. We also took a short look at, how we can easily take advantage of more thread by using the OpenMP API. There are plenty more implementations for using thread in C. But we've focused on the main concepts of threads and how to handle problem that can occur by using threads. you should now be able to experiment with diffrent Implementations and have a understanding how threads work.

## 5 Literature

- [1] Blaise Barney. Posix threads programming. <https://computing.llnl.gov/tutorials/pthreads/>.
- [2] Many. Thread (computing). [http://en.wikipedia.org/wiki/Thread\\_\(computing\)](http://en.wikipedia.org/wiki/Thread_(computing)).