Lesson 7

Christian Schwarz, Jakob Krebs
9.12.2019

Contents

Sources and Solutions

- we publish all code written in this course at `https://github.com/jkrbs/c_lessons`
- we will publish example solutions of the tasks on same site
- send us questions or your solutions to c-lessons@deutschland.gmbh

Function Pointers

Function Pointers

- As we know, a pointer is really just a memory address
- The code for functions is also in memory, so it also has an address.
- As long as different functions have the same argument types, we can call them using a function pointer:

```
1  int foo(int x, int y){ puts("called foo"); }
2  int foo(int x, int y){ puts("called bar"); }
3  // func_ptr is the newly created function pointer
4  int (*func_ptr)(int,int);
5  func_ptr = &foo; // we give it foo's address
6  func_ptr(1,2); // we call foo
7  func_ptr = bar; // the '&' is optional, think arrays
8  func_ptr(1,2); // we call bar
```

Since the above type declaration is slighly complicated, it's often a good idea to alias the function type:

```
1  // we can give the parameters names but it's optional
2  typedef int (*event_callback)(int event_id, void* context);
3
4  // now we can initialize a variable like this:
5  event_callback callback = &my_event_handler;
6  int result = callback(1, NULL);
```

Type Qualifiers

const

To give more information about a variable to the compiler, you can *qualify* its type.

The most common type qualifier is `const`. It prevents the qualified variable from being modified. If you try anyways you will get a compiler error.

```
1  // request that x can't be written to (after initialization)
2  int const x = 3;
3  x = 3; // error: assignment of read-only variable 'x'
4
5  void f(int *a); // forward declaration for f
6  f(&i); // warning: 'foo' [...] discards 'const' qualifier [...]
```

But this is C we're talking about, so of course there's a way: `*(int*)&x = 3;`

Compiles no problem. **But** what happens is undefined behaviour, so your progamm would no longer be valid.

From the west-const to the east-const

Normally, a qualifier refers to the type to its **left**, but the following is also valid (and more common!):

```
1  const int a;            // equal to 'int const a'
```

Watch complex types:

```
1  const int *foo;         // mutable pointer, constant integer
2  int const *foo;         // same as above
3  int * const foo;        // constant pointer, mutable integer
4  int const * const foo;  // everything constant
```

volatile

`volatile` prevents the compiler from doing aggressive optimizations on a variable. For example:

```
1  volatile bool interrupt_occured = false;
2  while (! interrupt_occured ){
3      // we don't change interrupt_occured , so the compiler
4      // might assume that it can optimize away the check
5  }
```

This is mainly used in low-level programming:

- Hardware access (memory-mapped I/O)
- Threading (another thread modifies a value) (**!!** be very careful here)

volatile example C

```c
#include <stdio.h>

int main(void) {
    int i = 42;
    printf("%d\n", i);
}
```

```c
#include <stdio.h>

int main(void) {
    volatile int i = 42;
    printf("%d\n", i);
}
```

volatile example assembly

After compilation with *gcc -O3*:

```
.LC0:
    .string "%d\n"
main:
    sub     rsp, 8
    mov     esi, 42
    mov     edi, OFFSET FLAT:.LC0
    xor     eax, eax
    call    printf
    xor     eax, eax
    add     rsp, 8
    ret
```

```
.LC0:
    .string "%d\n"
main:
    sub     rsp, 24
    mov     edi, OFFSET FLAT:.LC0
    xor     eax, eax
    mov     DWORD PTR [rsp+12], 42
    mov     esi, DWORD PTR [rsp+12]
    call    printf
    xor     eax, eax
    add     rsp, 24
    ret
```

The compiler could not pass 42 to printf directly once we made i volatile .

restrict

Restrict guarantees to the compiler that nobody else is writing to the memory of a pointer (the pointer is not aliased). Therefore the compiler might do an optimization like this:

```
1  void f(char *restrict p1, char *restrict p2) {
2      for (int i = 0; i < 50; i++) {
3          p1[i] = 4;
4          p2[i] = 9;
5      }
6  }
7  // optimized version, only valid if p1 and p2 don't overlap
8  void f(char *restrict p1, char *restrict p2) {
9      memset(p1, 4, 50);
10     memset(p2, 9, 50);
11 }
```

Since this is purely an optimization, restrict never changes the output of a valid program.

Parallelism

Executing code in parallel

Each program has a process associated with it. At program start, this process has exactly one thread executing your `main` function.

To achieve parallelism, you can

- create a new process running the same code
- call a function in a new thread

In Unix systems, processes are created with the `fork` system call.
The new process will have its own memory to work with.
For starting threads, libraries such as `p[osix]threads` are used.
All threads of a process share the same memory.

Use the fork

```
1  #include <unistd.h>
2  int main(void) {
3      pid_t pid = fork();
4      if (pid == 0) {
5          /* do stuff in child process */
6      } else if (pid > 0) {
7          /* do stuff in parent process */
8      } else {
9          /* fork failed */
10         return 1;
11     }
12     return 0;
13 }
```

Have a look at `man 2 fork` for further information.

pthreads

pthread_create

To execute a function in a new thread, use:

```
1  int pthread_create(pthread_t *thread,
2                     const pthread_attr_t *attr,
3                     void *(*start_routine) (void *),
4                     void *arg);
```

where

- `* thread` is where the thread's id will be stored
- `* attr` contains attributes for the thread (pass `NULL` for default)
- `start_routine` is the function to execute. Both the single argument and the return value must be `void *`.
- `arg` is passed to the function to be used as an argument

code example: threads

```c
1  #include <pthread.h>
2  #include <stdio.h>
3
4  void *hello_thread(void *tid) {
5      printf("Hello, I am thread %d\n", *(int *) tid);
6      pthread_exit(NULL);
7  }
8
9  int main(void) {
10     pthread_t threads[5];
11     for (int i=0; i < 5; ++i) {
12         if (pthread_create(&threads[i], NULL,
13                            hello_thread, (void *) i))
14             return 1;
15     }
16     return 0;
```

14

How threads end

- `pthread_exit` is called

```
1  void pthread_exit(void *retval);
```

- `pthread_cancel` is called from another thread

```
1  int pthread_cancel(pthread_t thread);
```

- `exit` is called from any thread (ending the process)

Waiting for threads

To wait for a thread to finish, there is `pthread_join`

```
1  int pthread_join(pthread_t thread, void **retval);
```

The thread passed to `pthread_join` must be joinable. The default is joinable, but one can disable this.

code example: joinable threads

```
1   int main(void) {
2       pthread_t threads[5];
3
4       for (int i=0; i < 5; ++i) {
5           if (pthread_create(&threads[i], NULL,
6                                hello_thread, (void *) i))
7               return 1;
8       }
9
10      void *st;
11      for (int i=0; i < 5; ++i) {
12          if (pthread_join(thread[i], &st))
13              return 1;
14          printf("Thread %d finished with %d\n", i, *(int *) st);
15      }
16
```

17

Mutual Exclusion

Mutexes

Threads can communicate with each other by manipulating global variables or the value behind the `arg` pointer we pass to `pthread_create`.

To avoid race conditions, the pthread library provides mutexes.

```
1 int pthread_mutex_destroy(pthread_mutex_t *mutex);
2 int pthread_mutex_init(pthread_mutex_t *restrict mutex,
3                 const pthread_mutexattr_t *restrict attr);
4 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

A mutex is a datatype that can be locked before and unlocked after accessing a variable.

```
1 int pthread_mutex_lock(pthread_mutex_t *mutex);
2 int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

mutex example

```
1  struct stuff {
2      unsigned a;
3      unsigned b;
4  }
5  struct stuff global = {1, 2};
6  pthread_mutex_t mutex;
7  //initialize the mutex
8  pthread_mutex_init(&mutex, NULL);
9  //[...]
10 void *thread(void *tid) {
11     pthread_mutex_lock(mutex);
12     global.b = a;
13     pthread_mutex_unlock(mutex);
14     pthread_exit(NULL);
15 }
```

Deadlocks incoming

```
1  void *thread_1(void *tid) {
2      pthread_mutex_lock(mutex_1);
3      pthread_mutex_lock(mutex_2);
4      /* do stuff */
5      pthread_mutex_unlock(mutex_1);
6      pthread_mutex_unlock(mutex_2);
7      pthread_exit(NULL);
8  }
9  void *thread_2(void *tid) {
10     pthread_mutex_lock(mutex_2);
11     pthread_mutex_lock(mutex_1);
12     /* do stuff */
13     pthread_mutex_unlock(mutex_2);
14     pthread_mutex_unlock(mutex_1);
15     pthread_exit(NULL);
16 }
```