Introduction to Operating systems

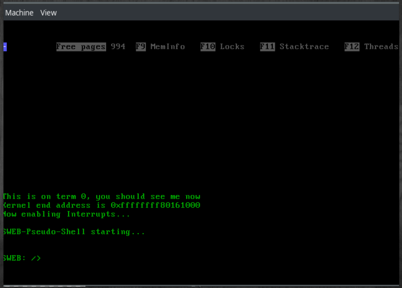
The Basics and the Code pitfalls

Christian Rieger

IAIK

01 Oktober 2024

It has been a long way



Topics for today

1. Setup and Build of SWEB
    - General Tooling (Compiler Qemu GDB...)
    - IDEs (graphical debugging)...
2. The SWEB structure (arch, common, userland...)
3. The first steps in the Operating systems course
    - Separation UserProcess/UserThread
    - Implement pthread_create with wrapper function
    - pthread_exit and pthread_cancel
4. POSIX? What is POSIX? how much do we need of this stuff?
5. Coding Standard and Coding Pitfalls
6. How to build good testcases?
7. Other Operating System Pitfalls

Setup and Build of SWEB

Basic tooling to build SWEB

Just use the tools provided by your distro

- What tooling do we need?
  - gcc, g++ and gdb (build-essential tooling should be sufficient)
  - qemu x86_64 for the main architecture
  - cmake for makefile generation
  - Other stuff (python, git, consider zsh + .oh-my-zsh)

Working/Debugging with IDE

Debugging with GUI makes finding bugs faster and easier

- What IDEs are supported with SWEB?
  - VisualStudioCode (free, fast lightweight IDE/editor, easy on the fly working experience)
  - CLion (Student version, debugging works)
  - Helix (terminal based, multimodal, lsp an dap support)

How to build SWEB?

From git to qemu, 6 simple steps to get SWEB running

- Clone your git repo into a folder on your laptop
  git clone <path to your repo>
- create a build folder, preferred on a ram disk
  mkdir /tmp/bs
- change directory in this folder
  cd /tmp/bs
- use cmake to generate the makefiles used to build SWEB
  cmake <wherever your git repo is>
- build the SWEB source code with parallel build
  make -j
- now its time to start qemu
  make qemu

The SWEB folder and source structure

Main Folders

- **arch** This folder contains the Architecture specific stuff
  - **arm** all the arch stuff for arm32 based platforms (rpi2)
  - **armv8** all the arch stuff for aarch64 based platforms (rpi3)
  - **x86** contains the architecture specifics for x86_32 and x86_64
- **common** contains all the platform independent code for the operating system
  - **console** all the code to handle input/output related tasks
  - **fs** contains the code for handling filesystem abstractions
  - **kernel** contains the core source code (UserProcess/UserThread)
  - **mm** KernelMemoryManager and PageManager
  - **ustl** Our friend and helper the ustl code (you know lists, maps vectors and stuff)
  - **util** All other stuff, but not very important...
- **userspace** This folder contains all the code for the user
  You will do your testcases and libc implementations in there

Our first steps in the OperatingSystems course

1. Separate UserProcess from the UserThread
2. Create thread context map and thread-handling
3. Implement pthread_create with a wrapper entry function
4. Implement pthread_exit and use it un the wrapper function
5. Implement pthread_cancel
6. Implement correct exit function (correct cleanup of all the threads)

POSIX and why we need it.

POSIX is a standard which defines the interfaces for the user.
It is used by Linux or Unix based operating systems.
You have to implement all tasks (at least the mandatory ones) according to this standard, so we have predefined interface guidelines we can relate to. So before starting with the implementation of a certain function like pthread_create you need to understand which behavior is defined by the POSIX standard. So make sure to correctly implement all arguments and return cases.

Attention!!! In the linux man pages there are a series of error-codes defined, but this does not mean you have to return those codes. These codes are stored in the *errno* variable. So in operating systems course you **do not** have to care about those. All error cases should return -**1**. Do not get confused with the error codes!!!.
Just return -**1** in case of an error.

Coding standard and coding pitfalls

During the Operating systems VU you will write a lot of code. A LOT OF CODE...
Someone will read and grade your code. Do not treat the Operating system like another user-space app! SWEB is a piece of embedded code. An Operating System is particular sensible to Bugs.
So what can we do to get the implementations as nice and stable as possible so we will not be screwed in the second assignment?
Easy just use the following guidelines for coding, ustl and error handling.

No boxed code!!!

Wrong!!!

```
int do_important_stuff(char* some_ptr, char* some_ptr1, int some_index_smaller_21)
{
    if(some_ptr != NULL)
    {
        if(some_ptr1 != NULL)
        {
            if(some_index_smaller_21 < 21)
            {
                /* do important stuff here */
            }
        }
    }
    return 0;
}
```

No boxed code!!!

Correct!!!

```c
int do_important_stuff(char* some_ptr, char* some_ptr1, int some_index_smaller_21)
{
    if(some_ptr == NULL || some_ptr1 == NULL || some_index_smaller_21 < 21)
        return 0; /* or -1 if this is an error */

    /* do important stuff here */

    return 0;
}
```

No unfortunate variable accesses

Wrong!!!

```
void do_stuff()
{
    ((Userthread*)currentthread)->getParentProcess()->getOtherMember()->do_something();

    ((Userthread*)currentthread)->getParentProcess()->getMutex()->lock();

    int something =
        ((Userthread*)currentthread)->getParentProcess()->getOtherMember()->get_something();

    ((Userthread*)currentthread)->getParentProcess()->getOtherMember()->set_something(23
        * something);

    ((Userthread*)currentthread)->getParentProcess()->getMutex()->unlock();
}
```

No unfortunate variable accesses

Correct!!!

```
void do_stuff()
{
    UserProcess* parent = ((Userthread*)currentthread)->getParentProcess();
    Mutex* parent_mutex = parent->getMutex();
    OtherMember* other_member = parent->getOtherMember();

    other_member->do_something();
    parent_mutex->lock();

    int something = other_member->get_something();
    other_member->set_something(23 * something);

    parent_mutex->unlock();
}
```

The uSTL

The uSTL is a small standard library which includes most of the data-structures we love in C++.
But we have to take in mind: we are not dealing with a user-application here, so we have to change our behavior accordingly.
This will help to make the code more stable and reduce debugging effort.
With the correct use of the uSTL, bugs can be found much more quickly or avoided at all.

Example uMap

Maps will be used very often is this course because you will need a data-structure which provides you a value by a certain key. For example in the UserProcess you will need a map which contains all the UserThreads and the key will be the thread_id. The same thing with the ProcessRegistry, but with UserProcesses and PIDs.

Do not access a map like this: *some_map[some_key].do_stuff();*
We need to make sure the entry on the place *some_key* is valid before using it.

Small example how to correctly get an entry from the map

Correct!!!

```cpp
void UserProcess::pthread_exit(size_t thread_id)
{
    assert(currentthread->getTid() == thread_id);

    // we use auto because of the generic iterator type
    auto user_thread = thread_map_.find(thread_id);

    // the thread needs to be in the map otherwise its a bug
    // and we will dereference a invalid iterator. This results in strange page-faults
    assert(user_thread != thread_map_.end());

    // now we can do stuff with our iterator
}
```

Small example how to correctly iterate through a map and delete an entry

Correct!!!

```cpp
// be careful!! the ustl map behavior might differ here
for(auto it = some_map.begin(); it != some_map.end(); i++)
{
    if(it->get_some_value() == 5)
    {
        it = some_map.erase(it);
    }
}
```

How to build good test-cases

Good test-cases do not need visual confirmation, they have to run on their own...
How do we do that? With **assert(0)**
If the assertion fails, you know that the test-case failed.
You can still print debug info, but you should stick with the asserts. Good test-cases save a lot of time, because you can focus on implementing new features instead of constantly worrying if in the meantime something else broke...
Some Tips:

- Use one file for a whole set of tests (all the threading) and execute them all after you changed something.
- Create a function for each test-case. So you also can build fast combination tests.
- All userspace-apps are linked statically so the binary size will be quite large, the sweb virtual image is quite small, so you will need to keep the number of test-files to a certain limit...

Small example of a good test-case

Correct!!!

```c
void test_pthread_create_invalid_args()
{
    pthread_t dummy = 0;
    assert(pthread_create(&dummy, 0, 0, 0) == -1);
    assert(pthread_create(0, 0, &some_thread_function, 0) == -1);
    assert(pthread_create(0, 0, 0, 0) == -1);
}
```

Another example of a good test-case

```c
static void* dummy_arg_var = NULL;
void* dummy_test_th_func(void* attr)
{
    dummy_arg_var = attr;
    assert(attr == 0xAA55);
}

void test_pthread_argument()
{
    pthread_t dummy = 0;
    assert(pthread_create(&dummy, 0, &dummy_test_th_func, (void*)0xAA55) == 0);
    /* (this makes sure the new thread will be scheduled) */
    for(int index = 0; index < 10; index++){sched_yield();}
    assert(dummy != 0 && "The thread-id should not be 0 in this case");
    assert(dummy_arg_var == 0xAA55 && "The thread has not been executed");
}
```

Other stuff to take in mind

- Try to avoid getting a page-fault when holding a kernel lock (check user args before getting the lock)
- Make sure every thread ends its own live, to avoid thread crossing operations -> this leads to locking issues
- Try to implement everything within the correct context, so do not implement any of the threading stuff outside of the UserProcess for example
- Use the F9 - F12 keys for debugging, there are very helpful
- Take full advantage of the gdb debugger, it really will help you to find bugs faster
- Use do not create to much branches and always do sync-merges with the master
- rm -rf * in the /tmp/bs directory helps sometimes to :-)
- Important!!! use asserts in sweb, they will help you to avoid running into some strange behavior which is hard to debug
- Take good care everything is correct locked, race conditions are very annoying to debug.