#uni/semester3/Betriebssysteme/chapter27/chapter-notes

1. Thread Creation

➜ first thing we need is some kind of thread creation interface. In POSIX:

```
#include <pthread.h>
int
pthread_create(pthread_t       *thread,
         const pthread_attr_t *attr,
               void           *(*start_routine)(void*),
               void           *arg);
```

➜ first argument thread is a pointer to a structure of type pthread_t. We will use this structure to interact with the thread and thus we need to pass it to the pthread_create() to initialise it.

➜ attr is used to specify any attributes to this thread, like setting the stack size or information about the scheduling priority of the thread. An attribute is initialised with a separate call to pthread_attr_init(). In most cases the defaults are fine, so we simply pass the value NULL.

➜ third argument asks which function should this thread start running? This is a function pointer and tells us the following is expected: a function name (start_routine) which is passed a single argument of type void*.

This returns a value of type void* (a void pointer). If the routine instead required an integer argument it would look like this:

```
int pthread_create(..., // first two args are the same
                   void *(*start_routine)(int),
                   int  arg);
```

➜ If it instead took a void pointer as argument, but returned an integer, it would look like this.

```
int pthread_create(..., // first two args are the same
                   int (*start_routine)(void *),
                   void *arg);
```

➜ arg is the argument passed to the function where the thread begins execution.

➜ Why do we need all these void pointers? Because this way we can pass any type of argument and having it as a return value allows the thread to return any type of result.

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

typedef struct {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *args = (myarg_t *) arg;
    printf("%d %d\n", args->a, args->b);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    myarg_t args = { 10, 20 };

    int rc = pthread_create(&p, NULL, mythread, &args);
    ...
}
```

Figure 27.1: **Creating a Thread**

➜ Here we have two arguments packaged into a single type (myarg_t). The thread once created casts the argument to type it expects.

2. Thread Completion

➜ pthread_join() is used to wait for completion.

```c
int pthread_join(pthread_t thread, void **value_ptr);
```

➜ First argument is of type pthread_t and is the thread to wait for. This variable is initialised with the thread creation routine.

➜ Second argument is a pointer to the return value you expect to get back. Because the routine can return anything, which means a pointer to void. pthread_join() routine changes the value of the passed argument, so you need to pass a pointer to that value, not just the value itself.

```
1  typedef struct { int a; int b; } myarg_t;
2  typedef struct { int x; int y; } myret_t;
3
4  void *mythread(void *arg) {
5      myret_t *rvals = Malloc(sizeof(myret_t));
6      rvals->x = 1;
7      rvals->y = 2;
8      return (void *) rvals;
9  }
10
11 int main(int argc, char *argv[]) {
12     pthread_t p;
13     myret_t *rvals;
14     myarg_t args = { 10, 20 };
15     Pthread_create(&p, NULL, mythread, &args);
16     Pthread_join(p, (void **) &rvals);
17     printf("returned %d %d\n", rvals->x, rvals->y);
18     free(rvals);
19     return 0;
20 }
```

Figure 27.2: **Waiting for Thread Completion**

➜ In this code a thread is created, passed a couple of arguments via the myarg_t structure. For return types the myret_t structure is used. When thread is finished running, the main thread, which has been waiting inside of the pthread_join() routine, then returns and we can access the values returned from the thread, namely the content of myret_t.

➜ Alternately we can create a thread with no arguments , with NULL as an argument. The same for pthread_join when we don't care about the return value.

➜ If we want to pass a single value to a thread we don't have to package is as an argument. See next code:

```
void *mythread(void *arg) {
    long long int value = (long long int) arg;
    printf("%lld\n", value);
    return (void *) (value + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    long long int rvalue;
    Pthread_create(&p, NULL, mythread, (void *) 100);
    Pthread_join(p, (void **) &rvalue);
    printf("returned %lld\n", rvalue);
    return 0;
}
```

➜ Never return a pointer which refers to something allocated on the thread's stack!

```
void *mythread(void *arg) {
    myarg_t *args = (myarg_t *) arg;
    printf("%d %d\n", args->a, args->b);
    myret_t oops; // ALLOCATED ON STACK: BAD!
    oops.x = 1;
    oops.y = 2;
    return (void *) &oops;
}
```

➜ In this case oops is allocated on the stack of mythread. When the thread returns the value is automatically deallocated (that's why the stack is so easy to use after all!) and thus returning a pointer to a now deallocated variable will lead to bad results. Compiler may complain of this!

```
vladb@VladB ~/G/h/S/B/B/c27_thread_api (master)>
gcc -o thread_create_with_return_stack thread_create_with_return_stack.c
thread_create_with_return_stack.c:22:22: warning: address of stack memory associated with local
      variable 'rvals' returned [-Wreturn-stack-address]
    return (void *) &rvals;
                    ^~~~~
1 warning generated.
```

➜ The use of pthread_create() to create a thread, followed by an immediate call to pthread_join() is a pretty strange way to create a thread. To accomplish this very exact test, these is a simpler way to do it, namely a **procedure call**. Most of the times we create more than just one thread and wait for it to complete, otherwise there is no purpose in using threads at all.

➜ Not all code that is multi-threaded uses the join routine. A web server might create a number of worker threads, and then use the main thread to accept requests and pass them to the workers for indefinitely time. Such long lived programs may not need to use join. A parallel program that creates threads to execute a particular task in parallel, will likely use join to make sure all such work complete before exiting or moving to the next phase.

3. Locks

➜ Next useful set of functions are those for providing mutual exclusion to a critical section via **locks**.

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

➜ Locks are quite useful when you have a region of code that is a **critical section** and needs to be protected to ensure correct operation.

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

➜ If no other thread holds the lock when pthread_mutex_lock() is called, the thread will acquire the lock and enter critical section.

➜ If another thread does hold the lock, the thread trying to grab it will not return from the call until it has acquired the lock. Only the thread with the lock acquired should call unlock.

➜ The first reason why the code above sucks is that it **lacks of proper initialization**. Locks have to be properly initialised to ensure that they work properly when lock and unlock are called. One way to init a lock is:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

➜ This sets the lock to the default values. The dynamic way to do it is like this:

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

➜ First argument is the lock, and second is a set of attributes. (Read the man page bla bla bla). We usually use the dynamic method. When done with the lock pthread mutex destroy() should be called.

➜ Second reason why the code above sucks is that it fails to check for error codes when calling lock and unlock. By not checking error codes, multiple threads could enter a critical section. One way to solve this is to use wrappers:

```
// Keeps code clean; only use if exit() OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

Figure 27.4: **An Example Wrapper**

➜ More complex programs, which can't simply exit when somethings goes wrong, should check for failure and do something appropiate.

➜ There routines are also of interest:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            struct timespec *abs_timeout);
```

➜ They are used in lock acquisition. The try lock version of acquiring a lock returns failure if the lock is already held. The timed lock version acquires of acquiring a lock returns after a timeout or after acquiring a lock (whichever happens first). The timedlock version with a timeout of zero is like the trylock. Both should be avoided, but are certain situations where

they can be useful.

4. Condition Variables

➜ The other major component of thread library is the **condition variable**. It is used to signal between between threads, if one is waiting for another to do something before it can continue. There are two routines used for this:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

➜ To use a condition variable you also need a lock. When calling one of the above routines, this lock should be held.

➜ The first routine puts the calling thread to sleep and thus waits for some other thread to signal it. Typical usage:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

➜ In this code after init of the lock and condition (one can use pthread_cond_init() (and pthread cond_destroy()) instead of the static initializer PTHREAD COND INITIALIZER) the thread checks if a variable ready has been set to something other than zero. If not the thread simply calls the wait routine and goes to sleep until some other thread wakes it.

➜ The code in the other thread looks like this:

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

➜ When signaling and wenn modifying the global variable ready, we make sure to have the lock held. This ensures that we don't introduce a race condition in our code.

➜ The wait call takes a lock as a second parameter, while the signal call only takes a condition. The reason is that the wait call in addition to putting the calling thread to sleep, it also releases the lock. If not the other thread can't acquire the lock and wake the sleeping thread. Before returning after being woken, the pthread_cond_wait() re-acquires the lock.

➜ Waiting thread rechecks the condition in a while loop, instead of an if statement, which

is odd. Using a while loop is the simple and safe thing to do. The reason is that there are some pthread implementations that could wake up a waiting thread; in this case without rechecking, the waiting thread will think that the condition has changed even though it has not.

➜ Sometimes it is temping to use a simple flag between two threads, instead of a condition variable and associated lock:

```
while (ready == 0)
    ; // spin
```

The associated signaling code would look like this:

```
ready = 1;
```

➜ Don't do this. It performs poorly and spinning for a long time just wastes CPU cycles. Second it is error prone. It is surprisingly easy to make mistakes when using flags to synchronise between threads.

5. Compiling and Running

   ➜ For all the code above you need to include the header pthread.h

   On the link line you must also link with the pthreads library, by adding the -pthread flag.

```
prompt> gcc -o main main.c -Wall -pthread
```

6. Type man -k pthread on linux to get more information and to see over 100 APIs.

ASIDE: THREAD API GUIDELINES

There are a number of small but important things to remember when you use the POSIX thread library (or really, any thread library) to build a multi-threaded program. They are:

- **Keep it simple.** Above all else, any code to lock or signal between threads should be as simple as possible. Tricky thread interactions lead to bugs.
- **Minimize thread interactions.** Try to keep the number of ways in which threads interact to a minimum. Each interaction should be carefully thought out and constructed with tried and true approaches (many of which we will learn about in the coming chapters).
- **Initialize locks and condition variables.** Failure to do so will lead to code that sometimes works and sometimes fails in very strange ways.
- **Check your return codes.** Of course, in any C and UNIX programming you do, you should be checking each and every return code, and it's true here as well. Failure to do so will lead to bizarre and hard to understand behavior, making you likely to (a) scream, (b) pull some of your hair out, or (c) both.
- **Be careful with how you pass arguments to, and return values from, threads.** In particular, any time you are passing a reference to a variable allocated on the stack, you are probably doing something wrong.
- **Each thread has its own stack.** As related to the point above, please remember that each thread has its own stack. Thus, if you have a locally-allocated variable inside of some function a thread is executing, it is essentially *private* to that thread; no other thread can (easily) access it. To share data between threads, the values must be in the **heap** or otherwise some locale that is globally accessible.
- **Always use condition variables to signal between threads.** While it is often tempting to use a simple flag, don't do it.
- **Use the manual pages.** On Linux, in particular, the pthread man pages are highly informative and discuss much of the nuances presented here, often in even more detail. Read them carefully!