threads

thread creation

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *
(*start_function)(void *), void *arg);
// creates a new thread
// after a call to pthread_create(), a program has no guarantee about which thread
will next be scheduled
```

```
- thread: pointer to where the new thread's identifier will be
- attr: use `NULL`for default attributes
- start_function:
        - the new thread start execution by calling start_function
        - start_function must return `void *` and take `void *` argument
        - void * start_function(void *arg);
- arg:
        - arguments which start_function is called with
        - if multiple arguments are needed, `arg`should point to a structure
```

thread termination

A thread can terminate in one of the following ways:

- thread's start function performs a return
- thread calls `pthread_exit()`
- thread is canceled using `pthread_cancel()`
- any of the threads calls `exit()`, or the main thread performs a return (in the `main()` function), which causes all threads in the process to terminate immediately

pthread_exit

```
int pthread_exit(void *retval);
// terminates the calling thread
```

```
- retval: return value for the thread
```

> ⚠ **Warning**

if the main thread calls pthread_exit() instead of calling `exit()` or performing a return, the the other threads continue to execute

pthread_cancel

```
int pthread_cancel(pthread_t thread);
// requests a thread cancelation
// returns immediately, it does NOT wait for the target thread to terminate
// a thread can be cancelable or not (by default, new threads are cancelable) and
calcelation can remain pending until a cancellation point
```

thread_join

```
int pthread_join(pthread_t thread, void **retval);
// waits for the specified thread to terminate
```

```
- retval: if not-NULL, it receives the terminated thread return value (value
specified when the thread performs a return or calls `pthread_exit()`)
```

> ⚠ **Warning**
>
> detached threads cannot be joined - detached threads are automatically removed when they terminate, their return value does not matter
> a thread can be marked ad detached using `pthread_detach()`

pthread_self

```
pthread_t pthread_self();
// returns the thread ID of the calling thread
```

notes

compilation

```
gcc -lpthread -D_REENTRANT -Wall file.c -o executable
```

- `D_REENTRANT` is called implicitly

> ⚠ **Warning**
>
> **USING NON-REENTRANT ROUTINES WON'T WORK AS EXPECTED SINCE THEY USE COMMON STORAGE IN AND UNSYNCHRONIZED WAY**

reentrant vs. non-reentrant

non reentrant routines are **NOT** thread-safe!
how to identify them?

- use of global/static variables
- lack of synchronization mechanisms
- modification of values passed as reference

thread-unsafe	reentrant version
`strtok`	`strtok_r`
`asctime`	`asctime_r`
`ctime`	`ctime_r`
`gethostbyaddr`	`gethostbyaddr_r`
`gethostbyname`	`gethostbyname_r`
`inet_ntoa`	`(none)`
`localtime`	`localtime_r`
`rand`	`rand_r`

forking in a thread

only the calling thread in replicated in the child process

exec() in a thread

when any thread calls one of the `exec()` functions, the calling program is completely replaced

thread synchronization

mutexes	POSIX semaphores	condition variables
- mutual exclusion zones between threads (processes can also use them if shared memory is used) - similar to a binary semaphore, but the thread who locks the mutex must be the one to unlock it	- signal events across thread - count objects in a synchronized way	- allow a thread to block/notify others in any condition - semaphores are a kind of condition variables - the implicit condition is the semaphore value being grater than 0

POSIX mutexes

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t
*restrict attr);
// initialize a mutex with the specified attributes
```

```
- attr: if `NULL`, the attributes used are the default
```

```
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
// declares and creates a mutex with default attributes
```

```
- does not generate errors!
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
// performs a lock on a mutex
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
// performs an unlock on a mutex
```

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
// tries to performs a lock on a mutex
```

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
// destroys a mutex
```