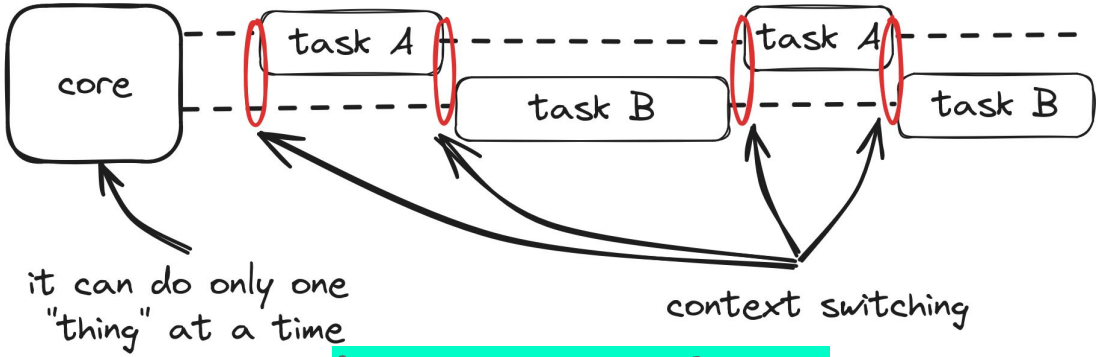
Concurrency

an introduction
by
Matteo Poggi

exact lab

What is Concurrency?

Ability carry out two or more actions in **progress at the same time**.



Saturday morning Example

Doing the housework :

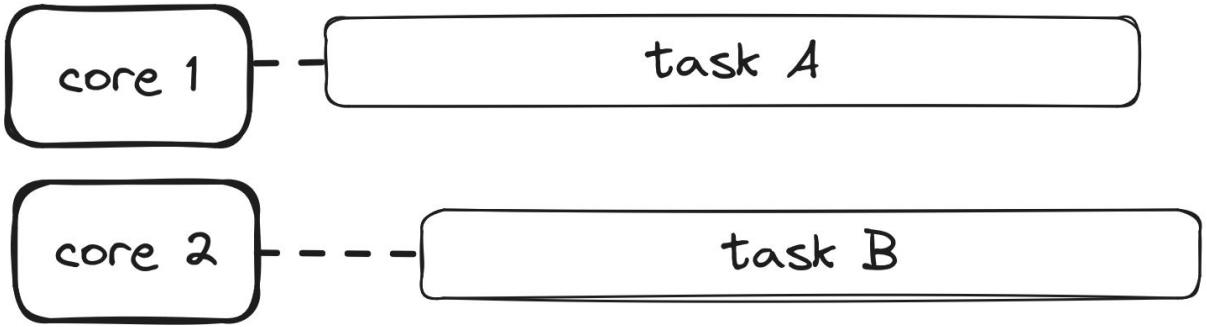
```
clean the kitchen
clean the bedroom
clean the bathroom
```

Doing the laundry:

```
use washing machine
use dryer
do the ironing
```

What about Parallelism?

Ability to execute two or more actions **at the same time**.



- Concurrency is about **structure**, parallelism is about **execution**.
- A concurrent program can be executed serially (on a single core).
- A concurrent program can be executed parallely (on multiple cores).

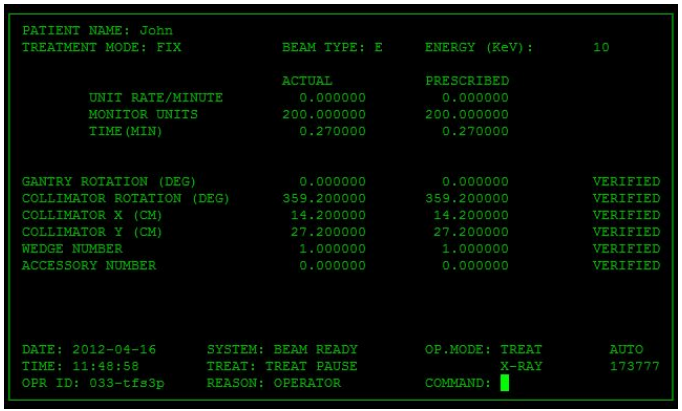
Two Forms of Concurrency

- **Multithreading** (lower level):
  several threads are spawned, they run concurrently (parallely if possible) on the CPUs at disposal. Synchronization among these threads is usually required to avoid data races and deadlock.

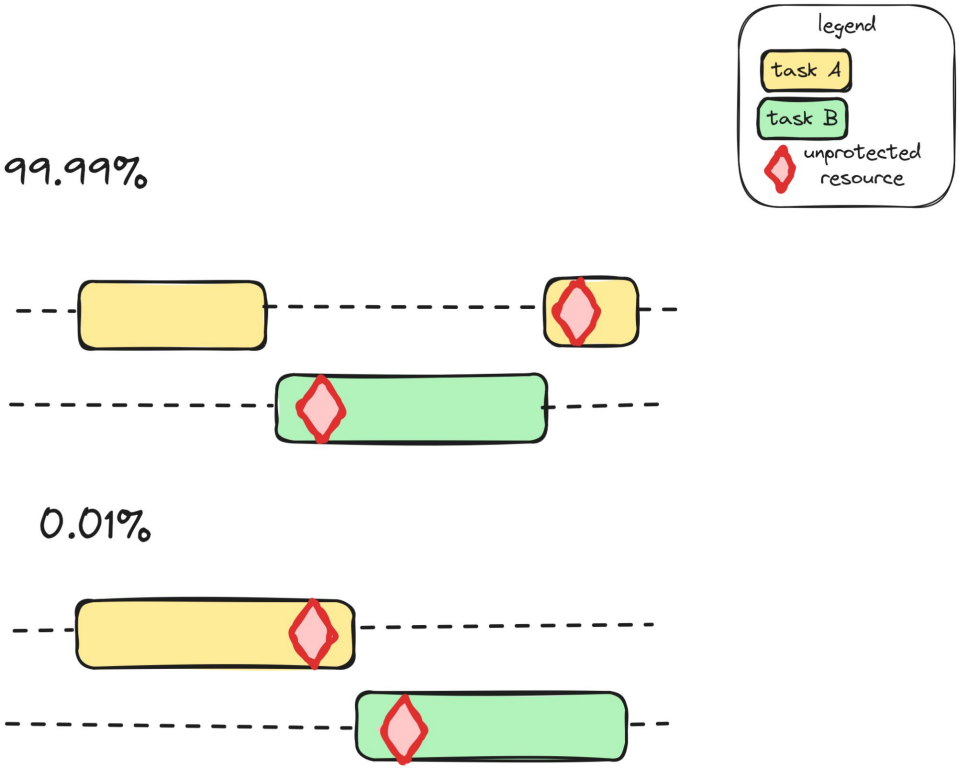
- **Asynchronous Programming** (higher level):
  tasks are started without waiting for the previous task completion. They can run in background, possibly in other threads. The use however has not to take care about these details.
  This can be achieved with callbacks, coroutines, promises and futures.

The Sorrows of Young MultiThreader

- Managing multiple threads requires synchronization among them.
- Wrong synchronization can cause:
  - race conditions;
  - deadlocks;
  - other undefined behaviors.
- Some wrong synchronization are very difficult to spot:
  - they can manifest infrequently (Threac25);
  - can be nearly impossible to replicate.



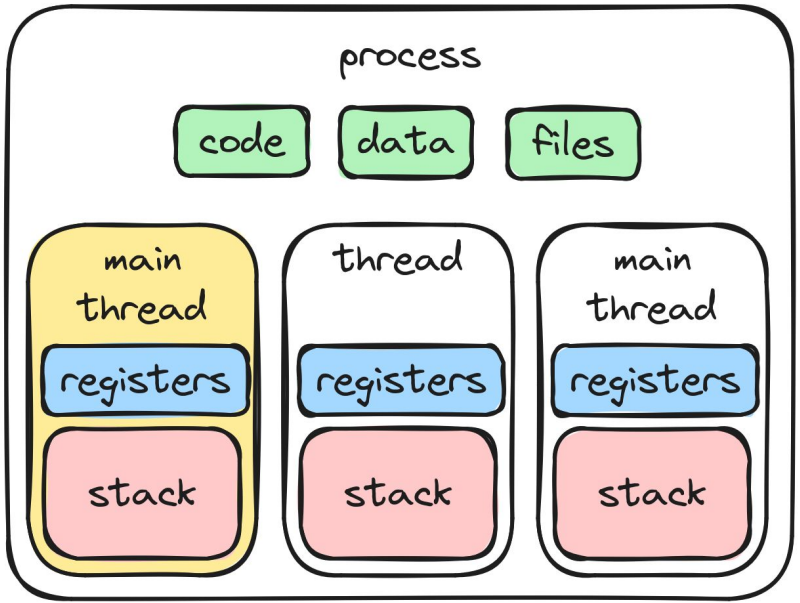
A Dangerous Situation

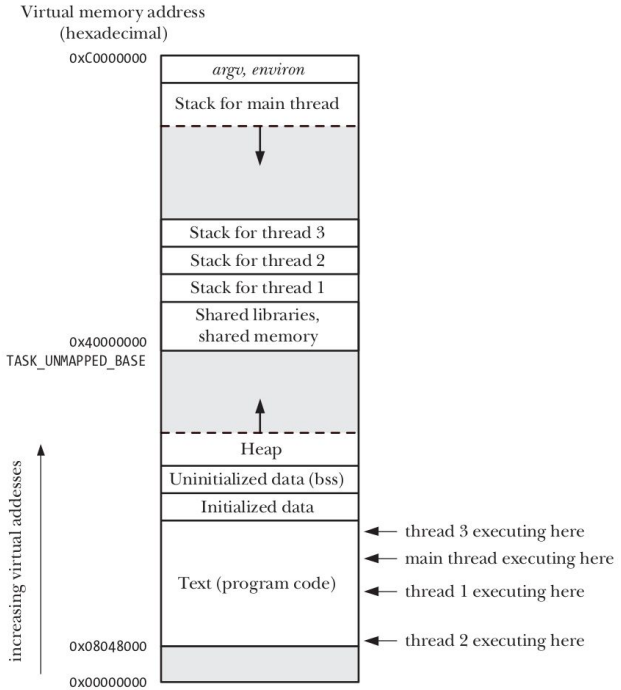


This is **not** a safe program:
if a data race is logically possible, even if it is unlikely it **must** be avoided by protecting shared resources.

Processes vs Threads

Both processes and threads can be used to achieve concurrency, but…



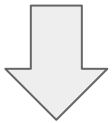


Fork a Process

```
#include <unistd.h>
#include <sys/types.h>

…

    pid_t pid = fork();
    switch (pid) {
    case -1:
        // something bad happened
        break;
    case 0:
        // child process
        break;
    default:
        // parent process
        break;
    }
```

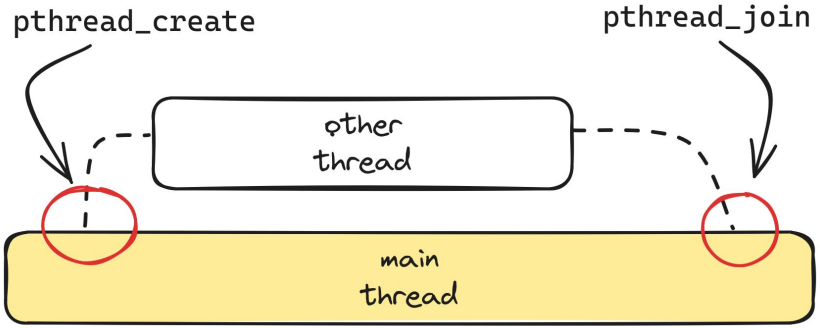
`fork()` creates the child process:

- Every process has a PID (`getpid()`)
- Every process has a PPID (`getppid()`)
- The virtual address space of the original process is copied-on-write
  (no copy occur as long as is read-only)
- After the copy, every process has its own virtual address space.



- relatively heavy/slow
- IPC can be a bit cumbersome
  (pipes, socket, message queues, shared memory)

Spawn a Thread




```
#include <pthread.h>

…

void* thread_function();

…

    pthread_t thread;
    if (pthread_create(&thread, NULL, thread_function, NULL))
        // something bad happened creating the thread

…

    if (pthread_join(thread, NULL))
        // something bad happened joining the thread
```

List Processes and Threads

- Using the commandP

  ```
  ps -efT
  ```

  one can see all Process ID (`PID`) and Thread ID (`TID` or `SPID`).
- Specifying

  ```
  ps -fT -p <PID>
  ```

  one can see the same data related to PID `PID`.
- Using top/htop the `PID` column refers actually to `TID`.

Examples: 1-Processes-and-Threads/*.c

A Note on Thread Libraries

We will use `pthread` library (Posix THREAD)

- this is available on Posix systems (LINUX, BSD, etc…)
- It is **not** available on Windows. Here we have `Win32` native libraries.

What about a portable library?

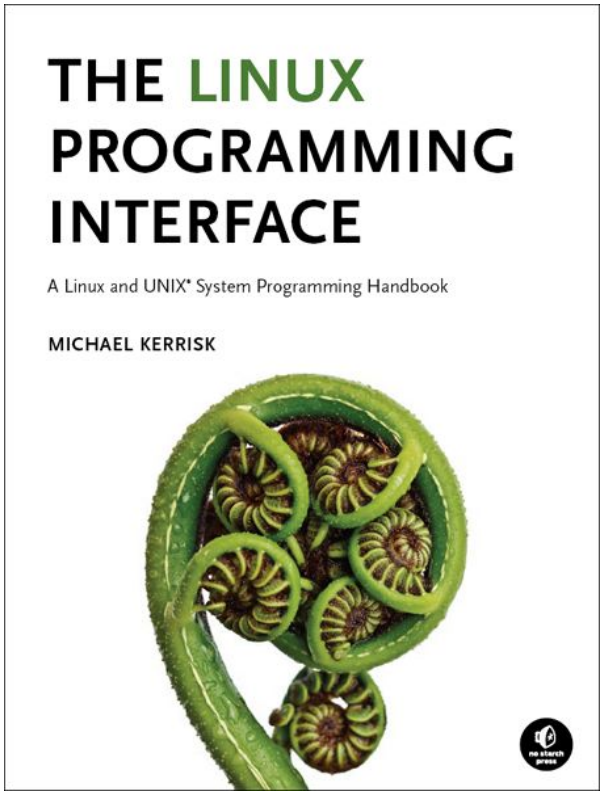
- in C++ (11 and later) is in the std (`<thread>` header).
  - It is capable of several thread programming paradigm (i.e. promise & future);
  - A lot of material can be found about it.
- in C (11 and later) there is as well however:
  - It had not been implemented on Windows (now it is);
  - very similar to `pthread`;
  - very few material about it.

Remarks on Pthread Library

- Unusual error codes `pthread_<do_something>` returns an `int`:
  - 0 for success;
  - POSITIVE for failure.

- Compile with option `-pthread`. This:
  - links programs with `libpthread`;
  - defines macro `_REENTRANT`.

A Must Have Reference



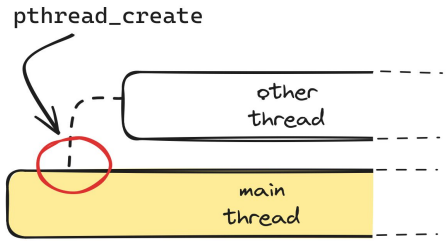

```
PIPE(2)                          Linux Programmer's Manual                          PIPE(2)

NAME
       pipe, pipe2 - create pipe

SYNOPSIS
       #include <unistd.h>

       /* On Alpha, IA-64, MIPS, SuperH, and SPARC/SPARC64; see NOTES */
       struct fd_pair {
           long fd[2];
       };
       struct fd_pair pipe();

       /* On all other architectures */
       int pipe(int pipefd[2]);

       #define _GNU_SOURCE             /* See feature_test_macros(7) */
       #include <fcntl.h>              /* Obtain O_* constant definitions */
       #include <unistd.h>

       int pipe2(int pipefd[2], int flags);

DESCRIPTION
       pipe()  creates a pipe, a unidirectional data channel that can be used for interprocess
       communication.  The array pipefd is used to return two file  descriptors  referring  to
       the  ends of the pipe.  pipefd[0] refers to the read end of the pipe.  pipefd[1] refers
       to the write end of the pipe.  Data written to the write end of the pipe is buffered by
       the  kernel  until  it is read from the read end of the pipe.  For further details, see
       pipe(7).

 Manual page pipe(2) line 1 (press h for help or q to quit)
```

Thread Creation

```
int pthread_create(pthread_t* thread,
                   const pthread_attr_t* attr,
                   void* (*start_routine)(*void),
                   void* arg)
```

- `pthread_t`: thread ID (implementation dependent, used as opaque);
- `pthread_attr_t`: thread attributes (pointer can be set to `NULL`);
- `*start_routine`: pointer to function for entry point of thread;
- `arg`: argument of `*start_routine` (pointer can be `NULL`).

NOTE: it returns immediately after thread creation.



Example: 2-Create-and-Join/1-no-Join.c

Thread Function Input and Output

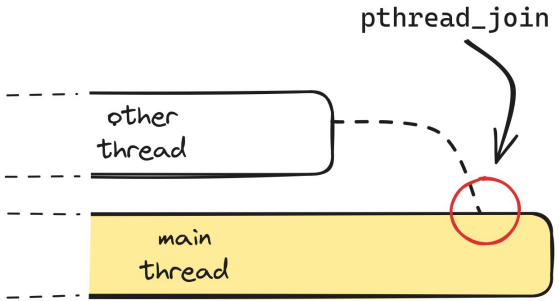
- Arg of thread function is a `void*` (type erasure):
  - Spawning thread allocates memory on heap and pass pointer (cast to `void*`) as argument: inside thread function it must be cast back to original structure;
  - Spawning thread use global variable (no need to pass global variable as argument);
  - Spawning thread pass some argument of type `T`, provided `size(T) <= size(void*)`: This is usually not portable and not compliant to strict ISO, but supported by most compilers.
- Return value of thread function is again a `void*`:
  - Return pointer to memory allocated inside thread function (not good idea…)
  - Return error code (again provided `size(T) <= size(void*)`, above caveats)
  - Return `NULL`, use other way to communicate.

It is possible to use function `pthread_exit(void* status)` instead of `return`.
This function terminates present thread even if is not called inside thread entry point function.

Examples: 2-Create-and-Joint/{5-Input-Output.c, 6-Input-Output-better.c}

Thread Joining

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



- `thread`: the thread ID;
- `retval`: the return `void*` value of the function;

It makes the caller wait for the `thread` to finish its execution.
Notice: any thread can be joined with any other thread in the process.

Examples: 2-Create-and-Join/{2-Master-Join.c, 3-Multiple-Threads.c}

Detaching a Thread

There is another possibility:

```
int pthread_detach(pthread_t thread)
```

The thread is no longer joinable. System does clean up automatically.

Often used in combination with

```
pthread_t pthread_self()
```

which return the `pthread_t` of the current thread.

Example: 2-Create-and-Join/4-Detach.c

Synchronization Issue

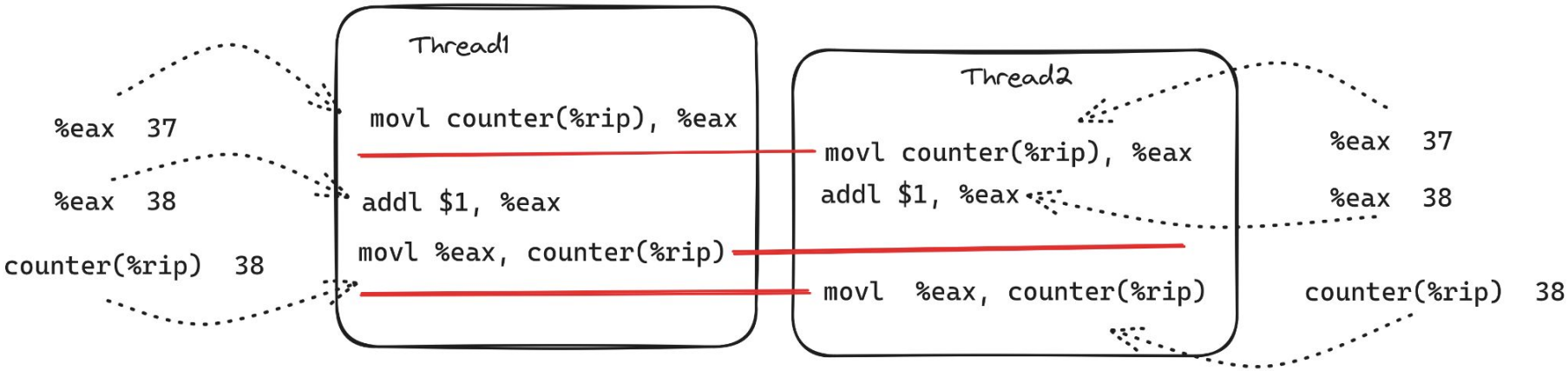
```
int counter = 0;     // global, shared variable

void* threadfunc(void*) {
     ++counter;
     return NULL;
}

int main() {
     …
     pthread_t thread1, thread2
     pthread_create(&thread1, NULL, threadfunc, NULL);
     pthread_create(&thread2, NULL, threadfunc, NULL);
     pthread_join(thread1, NULL);
     pthread_join(thread2, NULL);
     printf(“counter = %d\n”, counter);
     …
}
```

Examples: 3-Mutex/1-No-Sync.c

Thread Timeline

Assembly translation: (`gcc -S -fverbose-asm -g`)

```
movl    counter(%rip), %eax
addl    $1, %eax
movl    %eax, counter(%rip)
```



Atomic Solution

```
_Atomic int counter;
```

Several remark:

- Any type other thant function and arrays can be made `_Atomic`;
- Not every `_Atomic` type is guaranteed to be look-free (it may use a mutex);
- Defining a type `_Atomic` makes operations on it to be atomic; (example increment)
  beware: combination of atomic operation is **not** atomic!
- Atomic operations introduce synchronization between threads:
  5 memory order in C memory model
  (for most of them different threads will see different order of operation on a variable
  the default order `memory_order_seq_cst` preserve sequential consistency: global order)

Examples: 3-Mutex/2-Atomic.c

Mutex

Mutual **ex**clusion mechanism. It can be either locked or unlocked.

- Any thread can lock (or acquire) the mutex (only once)
  this is done before managing the shared resource
  - if the mutex is unlocked the current thread locks it and proceed;
  - if the mutex is locked (by another thread) the thread unlocking the mutex is paused until the mutex is released (by the same thread that acquired it).
- The thread that locked the mutex can unlock (or release) it
  this is done after having managed the shared resource
  other thread waiting on the mutex can now acquire it.

Any other behaviour (a thread acquiring multiple time the same mutex, releasing a mutex that has not been acquired) results in an UB.

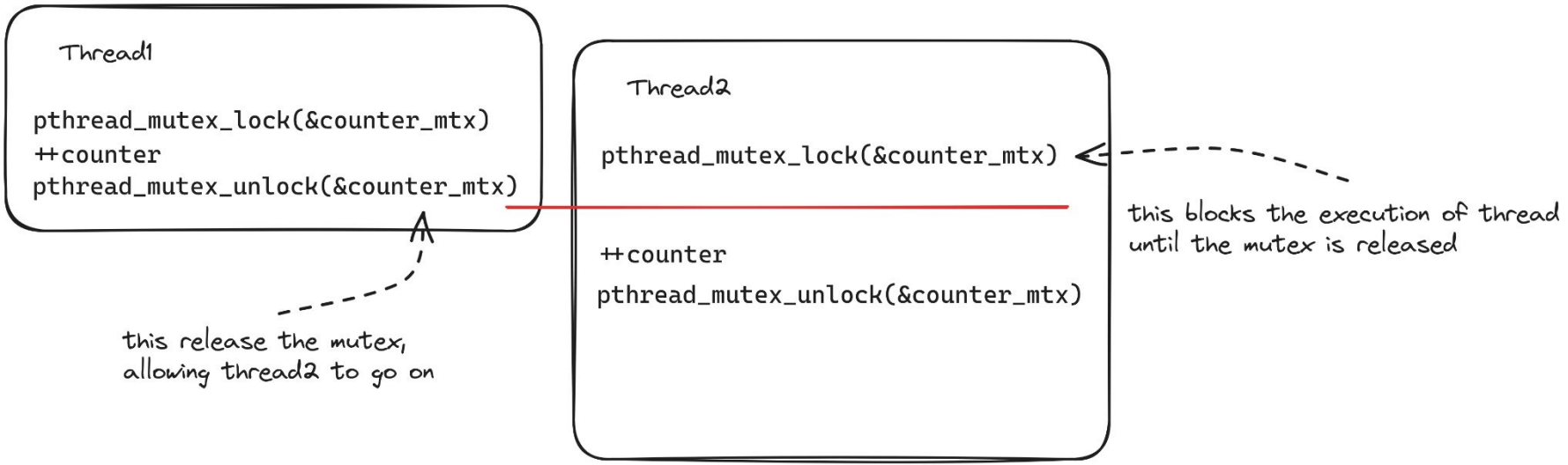
(but there are recursive mutex that may be acquired more than once by the same thread)

Use of Mutex

```
int counter = 0;
pthread_mutex_t counter_mtx = PTHREAD_MUTEX_INITIALIZER;

void* threadfunc(void*) {
    …
    pthread_mutex_lock(&counter_mtx);
    ++counter;
    pthread_mutex_unlock(&counter_mtx);
    …
}
```

Examples: 3-Mutex/3-Mutex.c

What is going on?



beware: is **up to you** to protect every access of the resource with a mutex:
if you forget to wraps an access of the resource with mutex lock/unlock you may still have a data race.

Mutex Function Call

Blocking calls:

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

Time-blocking call:

```
#include <time.h>

int pthread_mutex_timedlock(pthread_mutex_t *mtx,
                                        const struct timespec* abstime);
```

returns `ETIMEOUT` if it fails to lock the mutex due to a timeout.

Non-blocking call:

```
int pthread_mutex_trylock(pthread_mutex_t* mtx)
```

returns `EBUSY` if the mutex is already locked

Examples: 3-Mutex/{4-Timed-Lock.c, 5-Try-Lock.c}

What about CPU while Waiting for a Mutex?

Waiting for a mutex to be release is **not** busy waiting:

- with top you can see that the CPU consumption of the thread is irrelevant.
- in Linux it is implemented in terms of atomic variables and a subsystem called futex (fast userspace mutex) that manage context switching.
- due to these calls a mutex can have a certain impact on performances (~100 cycles).

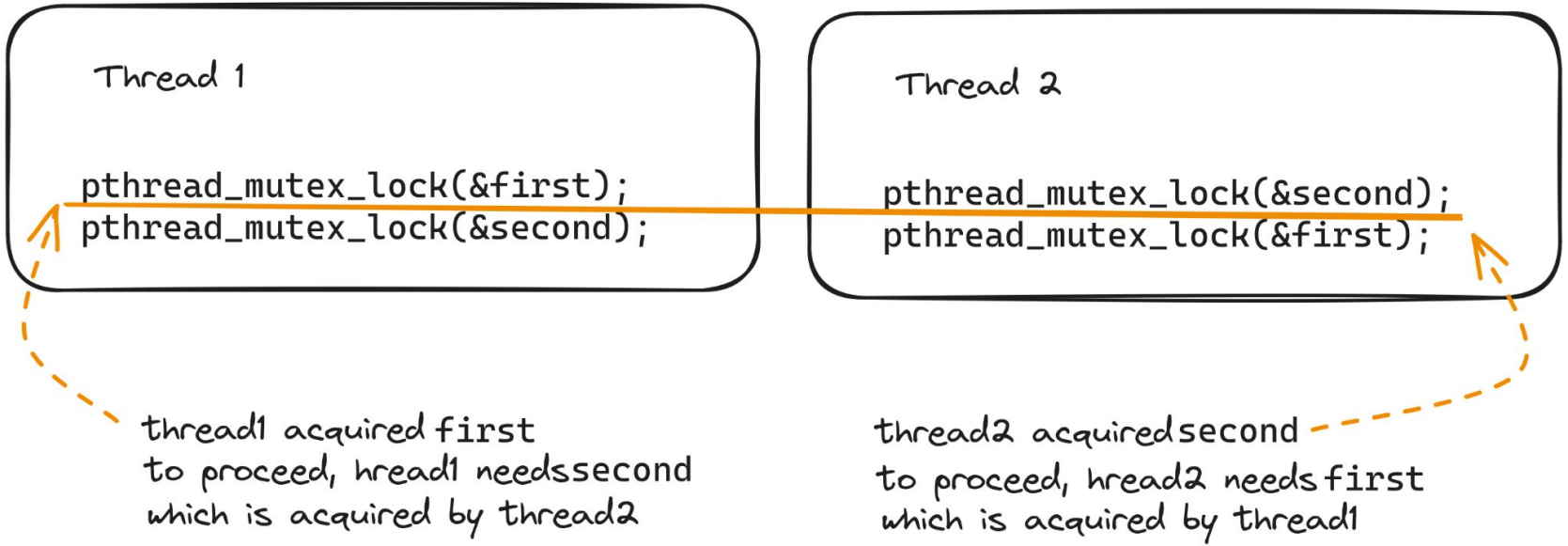
A spinlock is semantically similar to a mutex but **does** a sort of busy waiting:

- with top you can see the CPU consumption of the thread is ~100%
- lighter than a mutex (~just atomic test-and-set operation in a while loop)

Examples: 3-Mutex/{6-Busy.c, 7-not-Busy.c}

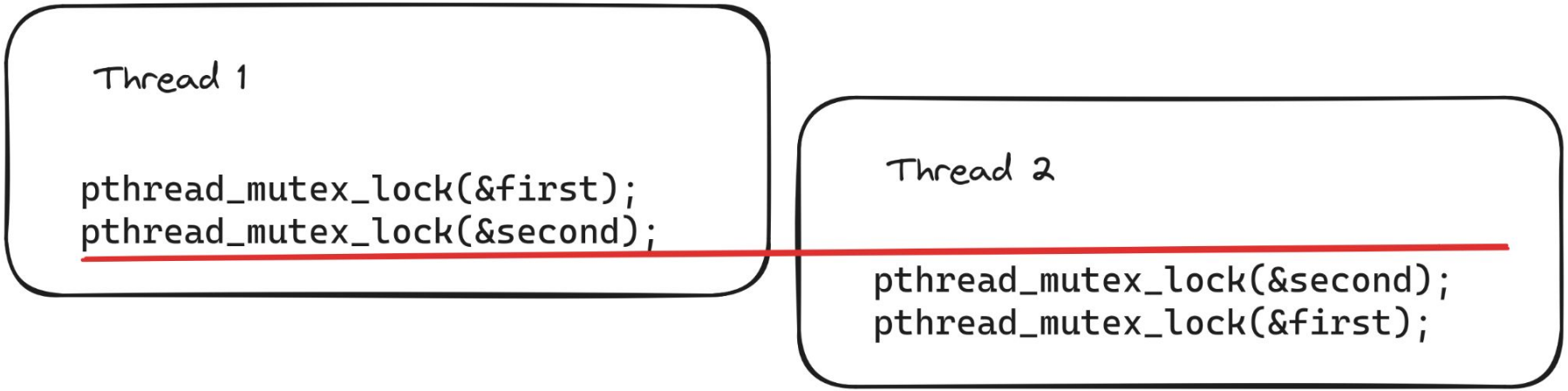
Mutex Deadlocks

Sometimes one need to lock two (or more) mutexes. A common situation is the



Example: 3-Mutex/{8-Deadlock.c}

Difficulty in Spotting Deadlocks



The source is **exactly the same** as before, but this time there is **no deadlock**.
This is because the deadlock does occur in certain lock orders.
It is likely that the deadlock will not arise often (however the program remains buggy).
To spot it: put some `sleep` between mutex locks and run the program several times

Deadlocks - How NOT to Fix

Sometimes one sees:

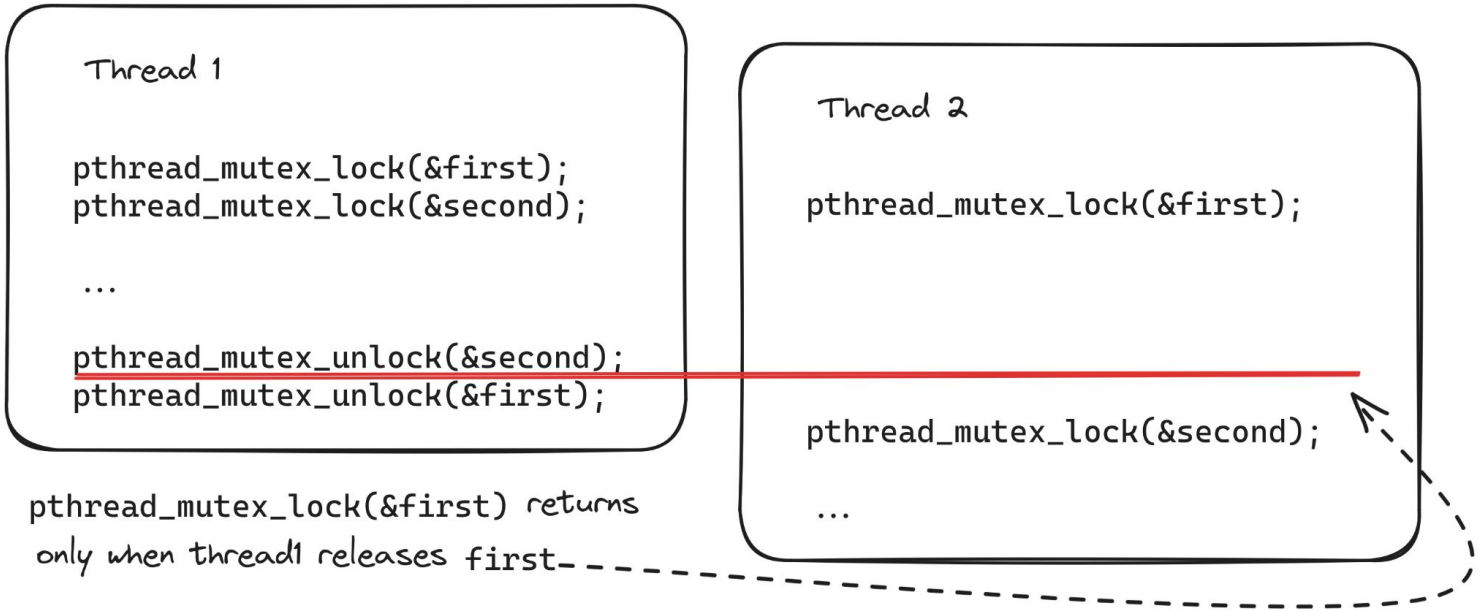
`pthread_mutex_lock(...)`  `pthread_mutex_timedlock(...)`

This transform a deadlock in a livelock.

Not a real solution.

Fixing Deadlocks - Mutex Hierarchy Strategy

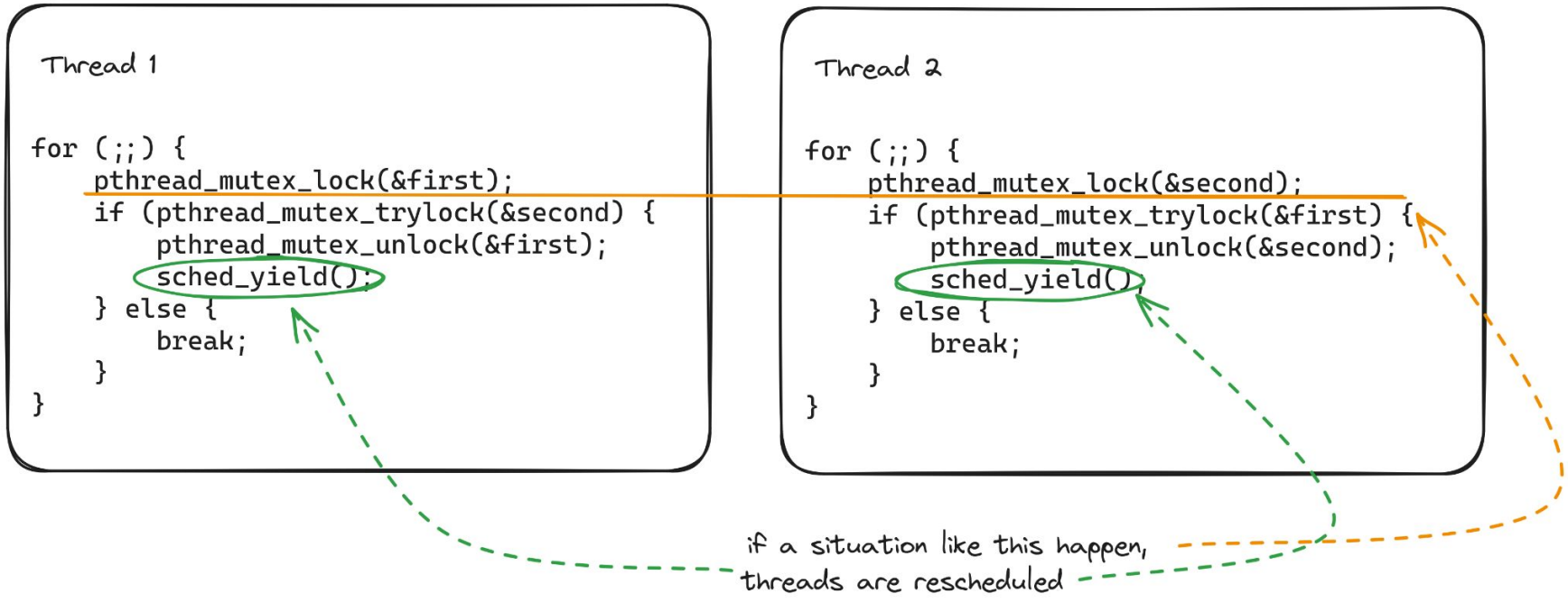
Every time we need to lock more than one mutex, we fix them in the same order.



Example: 3-Mutex/{9-no-Deadlock-Hierarchy.c}

Fixing Deadlocks - "Try and then Back off" Strategy

A more flexible (albeit less efficient) approach is the following:



Example: 3-Mutex/{A-no-Deadlock-Try-Backoff.c}

Mutex with Attributes and Dynamically Initialized Mutex

It is possible to specify som attribute for a mutex, in particular:

- `PTHREAD_MUTEX_ERRORCHECK`: enables check for trying to re-acquire a mutex (already acquired with the same thread) or to unlock a mutex not blocked by the thread.
- `PTHREAD_MUTEX_RECURSIVE`: enable the possibility for a thread to acquire multiple times the same mutex. The mutex must be unlocked the same number of times.

These mutexes are less performant than the normal one (`PTHREAD_MUTEX_NORMAL`)

To initialize a mutex with some attribute, o to initialize a mutex placed in the heap we cannot use the `PTHREAD_MUTEX_INITIALIZER`. We should use instead to initialize/destroy it

```
pthread_mutex_init(pthread_mutex_t* mutex);
pthread_mutex_destroy(pthread_mutex_t* mutex);
```

Thread Safety and Reentrancy

POSIX safety guarantees:
MT-Safe: function that can be executed in presence of other threads. (e.g.: printf, malloc);
AS-Safe: function that can be used in Asynchronous Signal (e.g.: getpid, getppid);
AC-Safe: function that can be used during Asynchronous Cancellation.

A reentrant function can be re-entered:

- a reentrant function does not act on global state variables
- a reentrant function does not use a mutex (can be re-entered by the same thread)

Producer - Consumer Model

The producer send data to the consumer via a channel (buffer);
The consumer, *as soon as* it receives data, elaborate them.

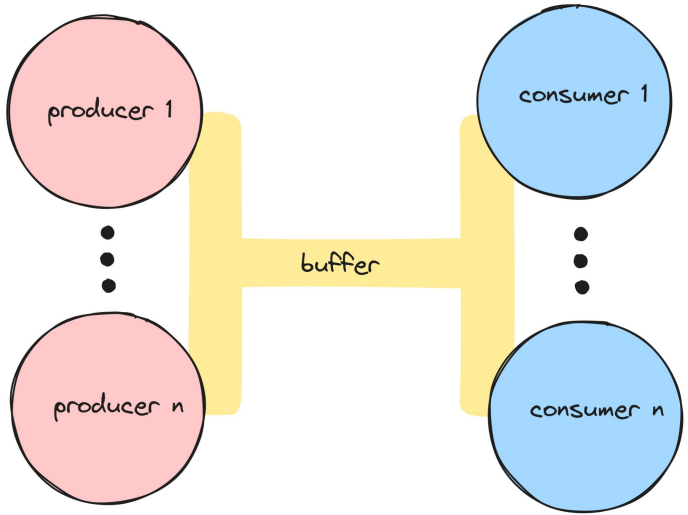
Questions:

- Is there some speed constraint between producer and consumer?
- How can the channel be implemented?
- How to synchronize producer and consumer?

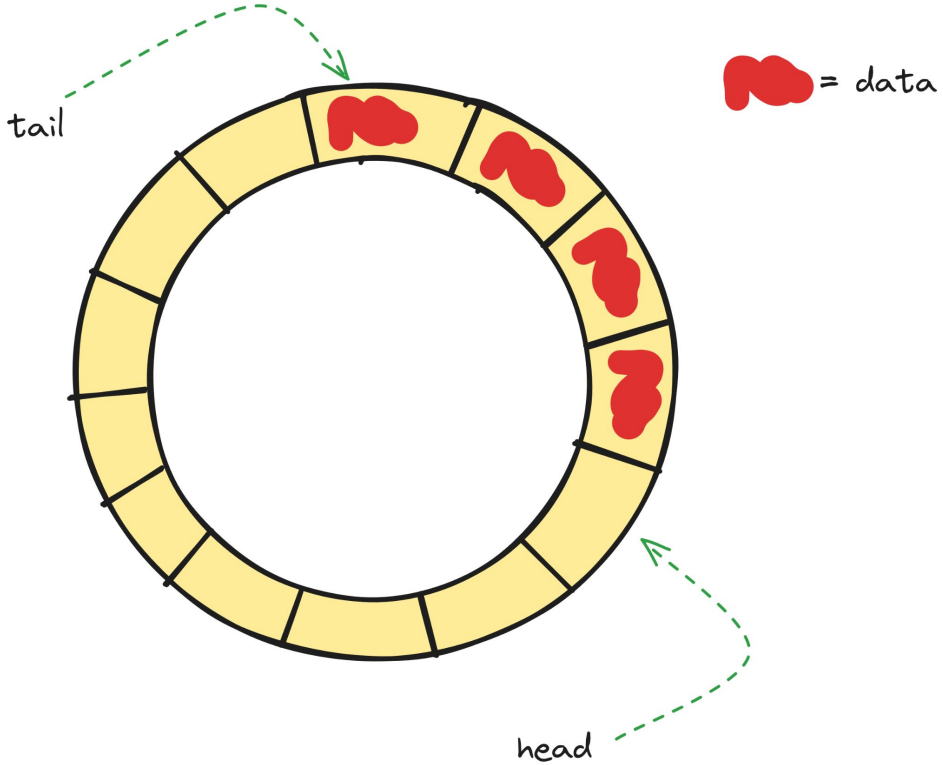
Multiple Producer / Consumer

Generalization of previous case:

- All producer and all consumer: same buffe
- Again, avoid race conditions;
- All producer and all consumer must be synchronized.



Homework - Circular Buffer



Circular Buffer - Interface

```
struct circbuf;
void cb_init(struct circbuf* cb,
             void* buf,
               size_t el_size,
             size_t capacity);
void cb_unset(struct circbuf* cb);
size_t cb_size(const struct circbuf* cb);
size_t cb_capacity(const struct circbuf* cb);
bool cb_push(struct circbuf* cb, const void* el);
bool cb_pop(struct circbuf* cb, void* el);

#ifdef DEBUG
ptrdiff_t cb_head_offset(const struct circbuf* cb);
ptrdiff_t cb_tail_offset(const struct circbuf* cb);
#endif // DEBUB
```

Polling Consumer

Using only mutexes one can do

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

while (!done) {
    pthread_mutex_lock(&mtx);
    … // do the job
    pthread_mutex_unlock(&mtx);
}
```

The consumer check continuously if there is something to do: busy waiting.

Example: 4-Condition-Variables/1-no-Condition-Variable.c

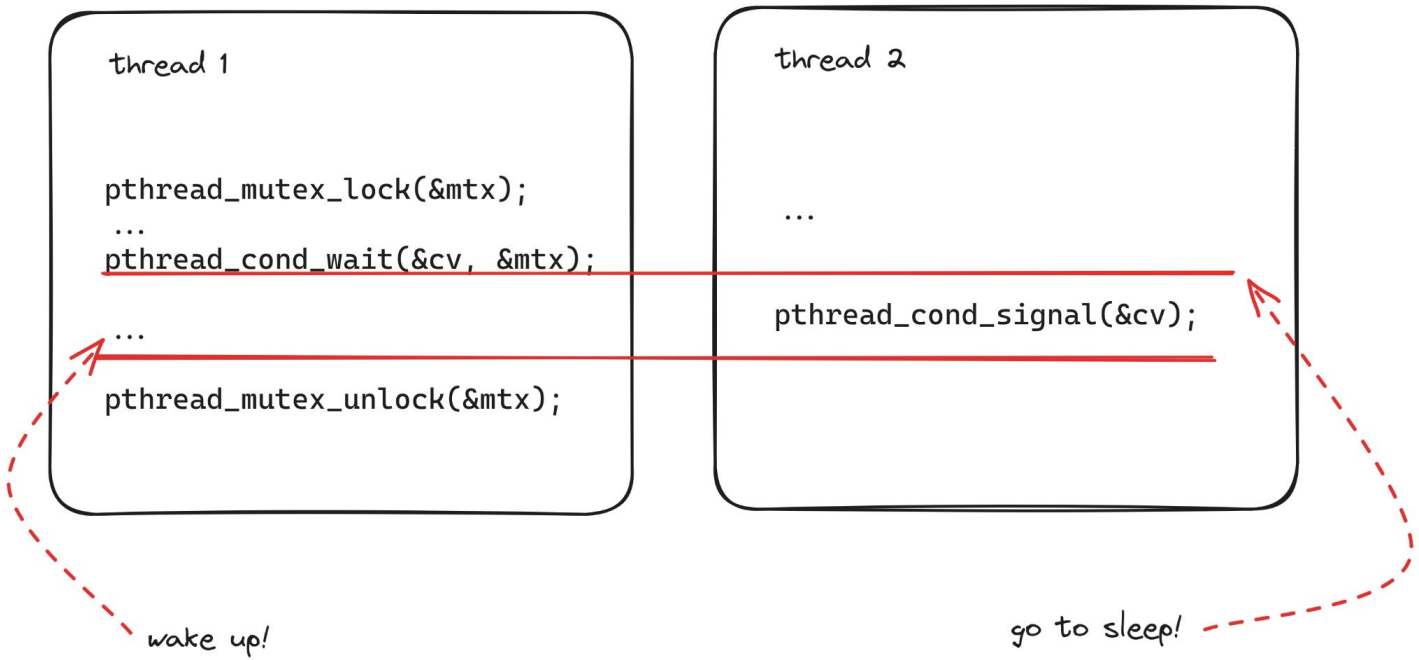
Condition Variable

They come in pair with a mutex. Two sets of instructions:

- pause instructions (wait, timedwait): pause the execution of the current thread, unlock the mutex until current thread is awaken;
- notification instructions (signal, broadcast): notify the sleeping thread(s) that something happened. Thread(s) waiting on the condition variable are awoken. Among them, a thread will acquire the mutex and go on with the execution.

NOTICE: the sleeping thread is **not** doing busy waiting!

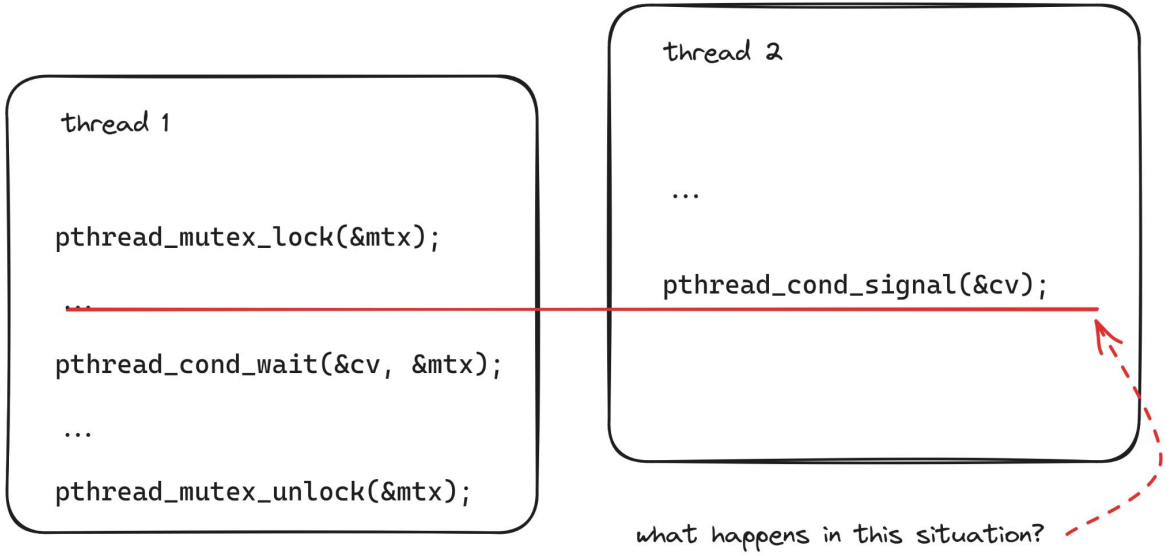
A not-so-good Condition



Spurious and Lost Wakeup

A thread waiting on some condition variable might be woken up even though no other thread actually notify the condition variable. This is called spurious wakeup.

A condition variable may be notified before a thread start waiting on it. In this case the wakeup is lost and a deadlock ensue.



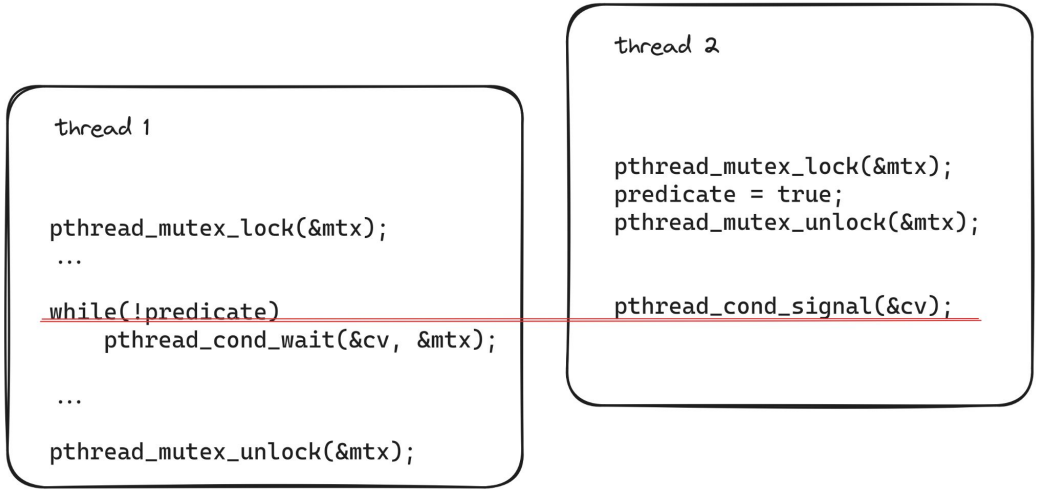
Example: 4-Condition-Variables/2-Minimal.c

Predicate

A way to solve both problem is to test for a predicate protected by the mutex.

If predicate is set to true (thread 2) *before* the while loop (thread 1) this thread does not even wait.

If a spurious wakeup does occur the predicate is tested again.



NOTICE: the predicate must be protected by the same mutex coupled to the condition variable.

Example: 4-Condition-Variables/3-Better.c

Condition Variables API

“Pause” functions:

```
int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mtx)
int pthread_cond_timedwait(pthread_cond_t* cond,
                           pthread_mutex_t* mtx,
                           const struct timespec* abstime)
```

“Notification” functions:

```
int pthread_cond_signal(pthread_cond_t* cond)
int pthread_cond_broadcast(pthread_cont_t* cond)
```

Static initialization is done via `PTHREAD_COND_INITIALIZER`,
for dynamic initialization and destruction: `pthread_cond_init` and `pthread_cond_destroy`.

Signal or Broadcast?

Signal function wakes up *at least one* thread waiting for the condition variable. The execution time of this call is faster, but if the predicate of the awoken thread is not satisfied we have a deadlock since no thread can proceed. This function has to be used when the predicate of all waiting thread are equivalent

Broadcast function wakes up *all* the threads waiting for the condition variable. The execution time of this call is slower, but it guarantees that each and every predicate of the waiting thread is checked.

Example: 4-Condition-Variables/{5-Bad-Signal.c, 6-Broadcast.c}