



**maze**<sup>™</sup>

**A development tool for concurrent programming**

# User Manual

**March 31, 2011**

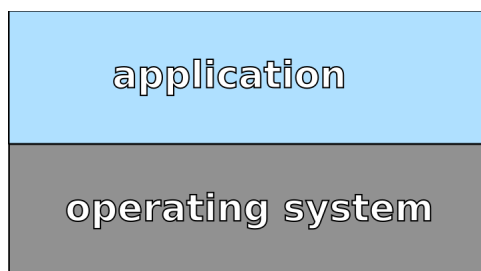
# Contents

<b>Overview</b> .....	<b>3</b>
<b>How maze works</b> .....	<b>4</b>
<b>Installation</b> .....	<b>5</b>
<b>Maze command line arguments</b> .....	<b>6</b>
The TEST mode.....	6
The RECONSTRUCTION mode.....	7
Reference.....	7
<b>Examining the results</b> .....	<b>8</b>
Application output.....	8
Maze output.....	11
<b>Examples</b> .....	<b>13</b>
Detecting a race condition.....	13
Detecting a deadlock.....	16

# Overview

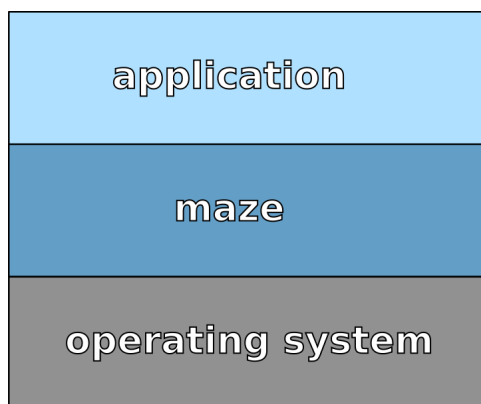
**Maze™ is a professional tool for parallel programming. It allows users to overcome the major difficulty in testing and debugging multiprocess and multithreaded applications: the lack of reproducibility.**

The behavior of a concurrent application is often unpredictable due to the non-deterministic nature of CPU sharing in a multitasking operating system. The operating system interleaves the execution of all existing processes. A user's program in general has no control over the process scheduling, which is done wholly by the system.



The schedule is affected by many different asynchronous events occurring in the system. As a result, the flow of a multithreaded application may vary from run to run. Because some inevitable bugs in such an application manifest themselves intermittently and are hard to reproduce, they remain undetected for a long time.

Maze is a simulator of the operating system scheduler. It creates a “layer” between the running application and the OS, taking full control over the scheduling of all threads and processes in the application.

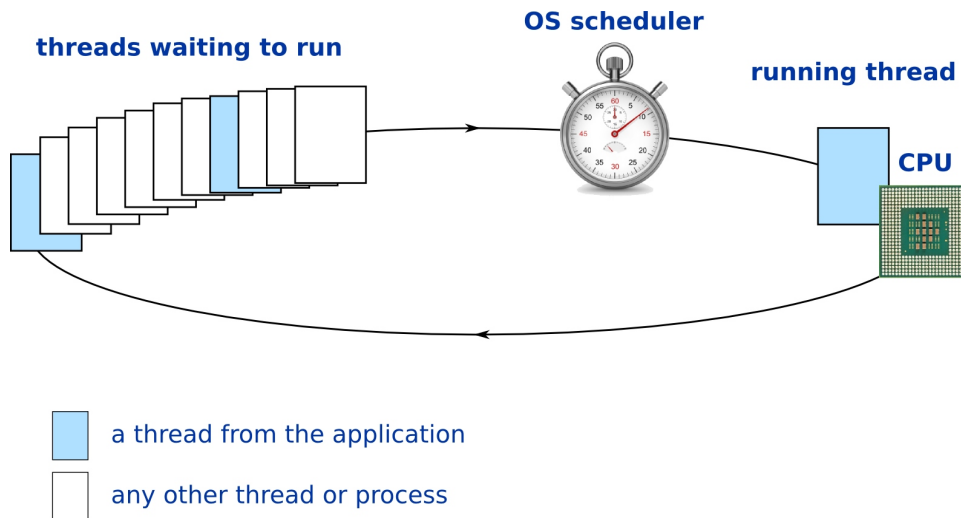


Running a user application repeatedly, maze generates various thread scheduling scenarios, similar to scenarios that might happen in the “real world”. If a bug is detected during a run, maze can reproduce the application behavior exactly at any time.

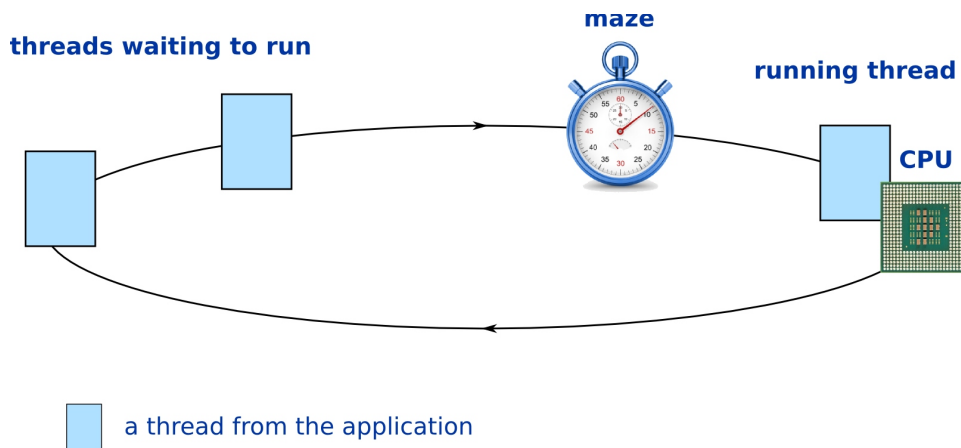
# How maze works

The only difference between a process running on its own, and the process running under maze is the way in which it is scheduled relative to other processes and threads within the same application.

In the former case the schedule is affected by the number, as well as states and priorities of all processes currently running on the machine. This schedule cannot be controlled by a user, is not recordable, and is not possible to reproduce.



When the process is controlled by maze, however, the schedule is not affected by any processes unrelated to the application. The schedule is deterministic, and it can be reproduced on request. If the process creates a child process or a thread, maze automatically takes control over the new process as well. All processes and threads of the application run, wait, or sleep by following directives from maze.



Maze can run an application multiple times, each time generating a unique thread/process scheduling scenario. This allows users to stress-test the application, and catch hard-to-reproduce race-conditions, deadlocks, and segmentation faults as long as enough schedule permutations have been tested. This mode of operation is called "TEST mode".

Another mode of operation is called “RECONSTRUCTION mode”. In this mode maze runs the application once, reproducing the scheduling scenario of any single run from the TEST mode session.

---

### Example:

The command

```
$ maze -r 1000 foo
```

starts the TEST mode session, running a program called `foo` repeatedly 1000 times. The subsequent command

```
$ maze -R 791
```

starts maze in the RECONSTRUCTION mode. Maze runs `foo` once, using the same thread scheduling scheme as it used during run #791 in the last TEST mode session, thus reproducing the program behavior, output and computation results obtained during this run.

---

Maze is easy to use. It requires no instrumentation of the binary. There are no source code requirements for the application<sup>1</sup>, the binary may be optimized, and the debugging information in the binary is optional<sup>2</sup>.

## Installation

Log into your account on <http://kloobok.com>, and download the latest version of `maze`. The binary is compressed; the name reflects its architecture and version; for example, `maze-64-bit-1.0-beta-2011-03-31.tar.bz2`.

Uncompress the binary

```
$ tar -xjf maze-64-bit-1.0-beta-2011-03-31.tar.bz2
```

`maze` is ready to run! We recommend saving `maze` binary in a directory listed in your `PATH`.

Verify that the version is correct:

```
$ maze -v
maze 1.0-beta-2011-03-31
```

---

<sup>1</sup> Stack printing is currently supported for C/C++ source code only.

<sup>2</sup> Maze does not need user binary debugging information in order to execute properly. It affects the stack printing only. Without debugging information function arguments and line numbers in the source code are not available.

## Maze command line arguments

Maze decides on its mode of operation based on the command line arguments. The order of the arguments does not matter. The arguments shown in brackets in this section are optional.

### The TEST mode

The TEST mode session starts with the following command:

```
maze cmd [ cmd ... ] [ -r runs ] [ -s seed ] [ -f ] [ -d dir ]
```

***cmd*** (“command”) has the '*prog-and-args*' [ *inst* ] format. There may be multiple *cmds*.

*prog-and-args* is the name of the application binary<sup>3</sup> with its own command line arguments. If the binary does not take any command line arguments, the quotes around *prog-and-args* are optional.

*inst* is a positive integer. It is an optional argument specifying the number of instances of the program to be run concurrently. The default value for *inst* is 1.

**-r *runs*** *runs* is a positive integer, specifying the number of runs in the TEST session. The default value for *runs* is 1.

**-s *seed*** *seed* is a positive integer, used as a seed for the maze random number generator. Different *seeds* result in different sets of process scheduling schemes generated by maze.

**-f** control both threads and child processes created with `fork()`. By default, maze controls only threads.

**-d *dir*** *dir* is a valid directory name. Maze stores the session data in the directory *dir/.maze/pid/*, where *pid* is the process id of the maze session itself. By default, *dir* is the run directory.

---

#### Example:

The TEST session with 3 shell commands ("ls -lt", "pwd", and "pwd") running concurrently 10 times can be initiated either with

```
$ maze 'ls -lt' pwd pwd -r 10
```

---

<sup>3</sup> If the name contains no slashes, maze follows the terminal shell lookup rules.

---

or with

```
$ maze 'ls -lt' pwd 2 -r 10
```

---

## The RECONSTRUCTION mode

The RECONSTRUCTION mode session starts with the following command:

```
maze [ -p pid ] [ -R run_num ] [ -d dir ]
```

**-p** *pid*        *pid* is the process id of the TEST session to be reproduced. By default, maze picks the most recent TEST session.

**-R** *run\_num*    *run\_num* is the run number to be reproduced. By default, *run\_num* is the last run in the TEST session.

**-d** *dir*        *dir* is a valid directory name. Maze looks for the TEST session data in the directory *dir/.maze/pid/*. By default, *dir* is the run directory.

---

### Example:

The RECONSTRUCTION mode session reproducing run #5 from the most recent of all TEST sessions stored under `/foo/bar/.maze` is initiated by

```
$ maze -R 5 -d /foo/bar
```

---

## Reference

The command

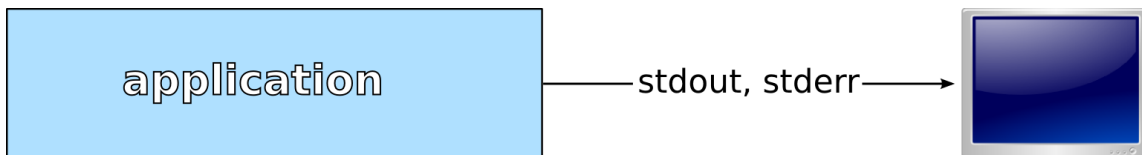
**maze -h**        prints the usage information described in this section;

**maze -v**        prints maze version.

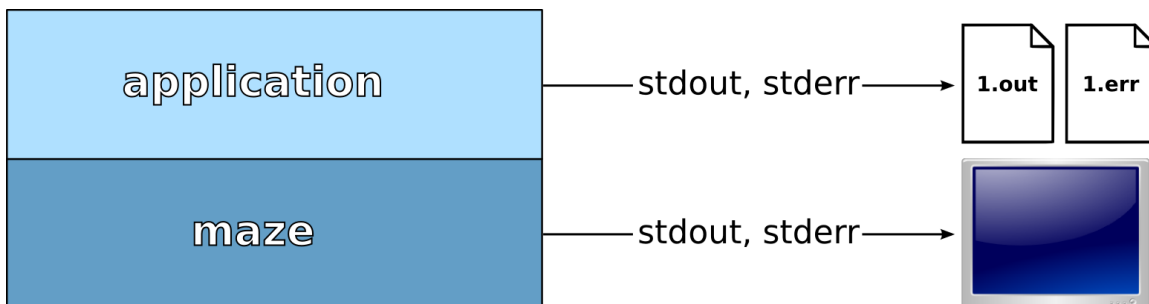
## Examining the results

### Application output

The standard output and standard error streams of a program are typically directed to the user's terminal:



However, when the program is run by maze, maze's own `stdout` and `stderr` are forwarded to the terminal. In order to avoid confusion, and to allow the user to examine the results from each run separately, maze redirects the application `std` streams to the files:



The files are named<sup>4</sup> after run numbers in the TEST mode: `1.out`, `1.err`, `2.out`, `2.err`, etc. They are stored in the directory `.maze/pid/`.

`.maze` is created under the run directory, unless the user specifies a different path<sup>5</sup>.

`pid` is the process id of the maze TEST session itself, *not* the process id of the application.

---

<sup>4</sup> There is no need to memorize the naming conventions described in this section. Maze always prints out a complete path to the application output files for every single run.

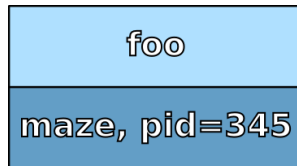
<sup>5</sup> The path is specified via "`-d dir`" command line argument.

---

Example:

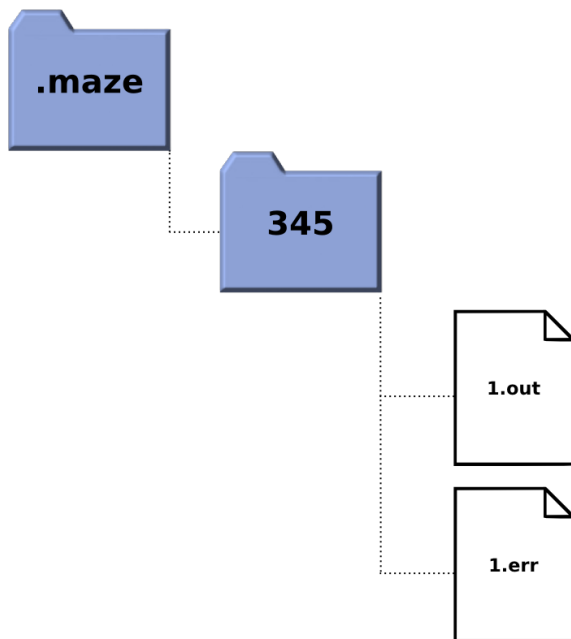
The maze TEST session with a single run of a program `foo` :

```
$ maze foo6
```



maze is running "foo" in TEST mode

results in the following directory tree:



maze creates a directory ".maze" unless it already exists

maze creates a directory ".maze/345" (345 is maze's own process id)

TEST results:  
stdout and stderr from "foo", run # 1

---

Maze in RECONSTRUCTION mode does not create a new directory. It stores the results in the directory created during the corresponding TEST mode session. The `stdout` and `stderr` streams are forwarded to the files `run_number.pid.out` and `run_number.pid.err`, respectively. `run_number` is the number of the TEST mode run being reproduced. `pid` is the process id of the maze RECONSTRUCTION mode session.

---

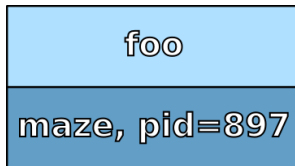
<sup>6</sup> Same as "`maze -r 1 foo`". The default number of runs in the TEST mode is 1.

---

Example:

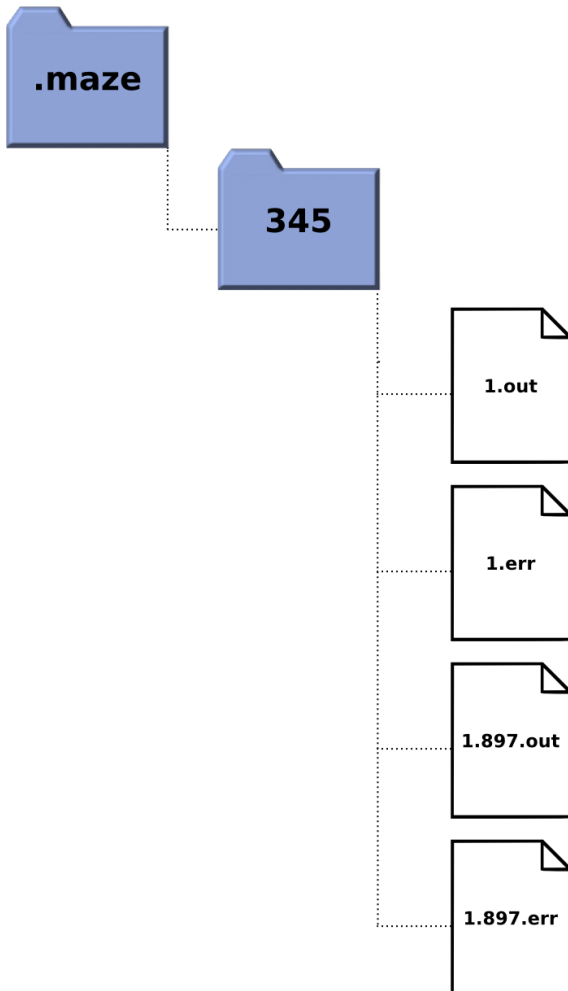
If we reproduce run # 1 from the previous TEST session:

```
$ maze7
```



maze is running "foo" in RECONSTRUCTION mode, reproducing run #1 from the session #345

maze will redirect foo's stdout and stderr into the files `.maze/345/1.897.out` and `.maze/345/1.897.err` respectively:



the directory ".maze" already exists

the directory ".maze/345" already exists

TEST results:  
stdout and stderr from "foo", run # 1

RECONSTRUCTION results:  
stdout and stderr from "foo", run # 1;  
"897" is maze's new process id

---

<sup>7</sup> Same as "maze -R 1 -p 345". By default maze in the RECONSTRUCTION mode reproduces the last run of the most recent TEST session.

## Maze output

Maze log of events starts with the header containing the information about maze itself. It follows with the information about current session:

- mode (TEST or RECONSTRUCTION) and number of runs

```
maze:Running in a test mode.  
maze:Number of test runs is 3.
```

- instructions for the future RECONSTRUCTION mode run (TEST mode only)

```
maze:A run from this session can be reproduced with the following command:  
maze:maze -p 5197 -R <run_index>  
maze:with run_index = 1,...,3
```

- run number and the TEST session id (RECONSTRUCTION mode only)

```
maze:Reproducing run # 2 from the maze test session with pid = 5197.
```

As maze proceeds with the runs, it prints this information:

- run number and full paths to the files where maze redirects the application's standard output and standard error streams

```
maze:Run # 1  
maze:  
maze:stdout > "/home/bar/.maze/5197/1.out"  
maze:stderr > "/home/bar/.maze/5197/1.err"
```

- process id(s) and command(s) of the process(es) started by maze

```
maze:The following process is running:  
maze:pid      command  
maze:5208     /home/bar/foo
```

Maze reports events occurring during the run:

- creation of new processes or threads<sup>8</sup>

```
A new thread (tgid = 5208, pid = 5209) was created by main thread (tgid =  
5208, pid = 5208)
```

---

<sup>8</sup> `tgid` is a “thread group id”; `pid` is a “lightweight process id”. Maze reports both values when identifying a thread. For a single-threaded process maze indicates `pid` only, since it is equal to `tgid`.

- processes or threads exiting

```
maze:Process (pid = 5209) exited normally.
```

- signals; if a signal is terminating the process, maze prints the process' stack

```
maze:Process (pid = 17388) received signal 6 (Aborted) from (pid = 17388).
```

- deadlocks<sup>9</sup>; maze prints a list of blocking processes and threads, their stacks and the blocking conditions

```
maze:ERROR: A deadlock was detected in the run # 30.  
maze:The following processes are blocking:  
maze:  
maze:Process (pid = 17325) is waiting for a mutex.  
maze:  
maze:Process (pid = 17326) is waiting for a signal (paused).
```

Maze log is forwarded to the terminal. If you choose to redirect the output to the file, note that maze log is printed into `stdout` stream, but the warnings and error messages, such as

```
maze:WARNING: Executable file "/bar/foo" has no debugging information.
```

are printed into `stderr`.

---

### Example:

```
$ maze foo 1>my_log 2>my_errors
```

The log is printed into the text file `my_log`. The file `my_errors` contains warnings and error messages (if any).

---

<sup>9</sup> Maze uses the term “deadlock” in more general sense than “all threads are waiting for mutexes”. The term describes any situation when all threads and processes in the application are blocking.

## Examples<sup>10</sup>

### Detecting a race condition

In this simple example the main thread creates two threads, waits while each thread increments the counter `count` by 1, and prints the final value of the counter.

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

static void * increment(void *);

int main()
{
    int count = 0;                                // initialize the counter

    const int num_threads = 2;
    pthread_t tid[num_threads];
    int error;
    int i;

    for(i = 0; i < num_threads; i++)             // create all threads
    {
        error = pthread_create(&tid[i], 0, &increment, (void *)&count);
        assert(error == 0);
    }

    for(i = 0; i < num_threads; i++)             // wait for every thread
    {
        error = pthread_join(tid[i], NULL);
        assert(error == 0);
    }

    printf("%d\n", count);                       // print the counter

    return 0;
}

static void * increment(void * count)
{
    *((int *)count)++;                          // increment the counter

    return NULL;
}
```

---

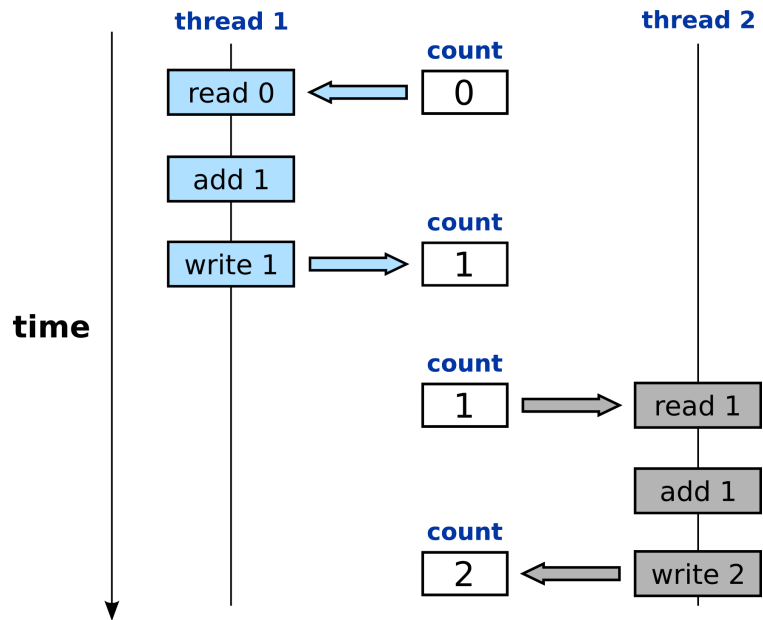
<sup>10</sup> The programs in the examples do not perform any meaningful work. They are just isolated test cases illustrating potential errors in a multithreaded code.

Compile the code and run it several times:

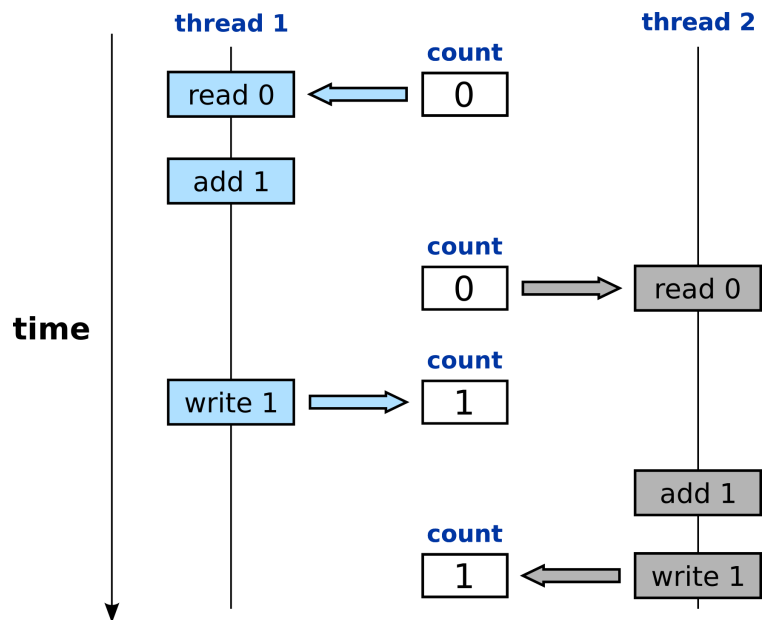
```
$ gcc -g race_condition.c -o race_condition -lpthread
```

```
$ ./race_condition
```

In all probability the value printed by `race_condition` is 2 every time. Indeed, while main thread is busy creating the second thread, the first thread has a plenty of time to increment the counter. By the time the second thread gets to the counter, its value is already 1.



Yet there is still a possibility that the second thread reads the counter *before* the first thread assigns it the incremented value. In this case the final value printed by `race_condition` must be 1.



Maze must be able to reveal this behavior.

Let maze test the program with 5000 TEST mode runs<sup>11</sup> and examine the output.

```
$ maze ./race_condition -r 5000 > /dev/null12 &
```

If maze *pid* was, for instance, 315, there are 5000 ASCII files in the directory `.maze/315/`: `1.out`, `2.out`, ..., `5000.out`, containing `count` values printed by `race_condition` during the corresponding runs.

Most of the files contain `2` – the most probable counter value. Check if any `count` values are different:

```
$ grep -v 2 .maze/315/*.out
```

The result may look like this:

```
.maze/315/1392.out:1
.maze/315/2613.out:1
```

In other words, in the runs #1392 and #2613 the final value of the counter was 1.

At this point a reasonable question to ask is “Why bother with maze, instead of simply running the program 5000 times?” The most important reasons are:

- In real life the thread scheduling is affected by the threads' priorities and the system-specific features, such as the number of CPU on your machine. This bias may further reduce the probability of the corner case events<sup>13</sup>. **Maze generates diverse and unbiased schedule.**
- If a process blocks (i. e. “hangs”) during a regular run, the user has to address the problem before proceeding further. This is not very practical, especially for the overnight regression runs. **Maze does not block when the application process does.** Instead, maze prints the diagnostic information<sup>14</sup> and proceeds with the next run.
- Even if a problem is detected during a “regular” run, there is no reliable way to reproduce it on request. **Maze can reproduce the exact behavior of the process during every single run.**

As a demonstration, reproduce run #1392 from the TEST mode session.

```
$ maze -R 1392 -p 315
```

The result is the same:

`.maze/315/1392.new_pid.out` is identical to `.maze/315/1392.out`, with the counter value equal to 1.

---

11 Even better approach is to run 2-3 parallel maze sessions with different random seeds (`-s seed`).

12 The log is irrelevant in this example, all we need to know is maze's process id.

13 For example, at fork, the Linux kernel tries to run the child process first.

14 See the next section “Detecting a deadlock”

## Detecting a deadlock

This example uses a “classic” deadlock between two threads. Each thread locks and then unlocks two mutexes. The first thread acquires the mutexes in the order “mutex\_1, mutex\_2”, while the second thread acquires them in the order “mutex\_2, mutex\_1”.

```
#include <assert.h>
#include <pthread.h>

static void * simple_thread(void *);

pthread_mutex_t mutex_1, mutex_2;

int main()
{
    pthread_t tid = 0;

    // initialize mutexes

    int error = pthread_mutex_init(&mutex_1, 0);
    assert(error == 0);

    error = pthread_mutex_init(&mutex_2, 0);
    assert(error == 0);

    // create a thread
    error = pthread_create(&tid, 0, &simple_thread, 0);
    assert(error == 0);

    error = pthread_mutex_lock(&mutex_1);    // acquire mutex_1
    assert(error == 0);

    error = pthread_mutex_lock(&mutex_2);    // acquire mutex_2
    assert(error == 0);

    error = pthread_mutex_unlock(&mutex_2);  // release mutex_2
    assert(error == 0);

    error = pthread_mutex_unlock(&mutex_1);  // release mutex_1
    assert(error == 0);

    error = pthread_join(tid, NULL);
    assert(error == 0);

    return 0;
}
```

```

static void * simple_thread(void * dummy)
{
    int error = pthread_mutex_lock(&mutex_2); // acquire mutex_2
    assert(error == 0);

    error = pthread_mutex_lock(&mutex_1);      // acquire mutex_1
    assert(error == 0);

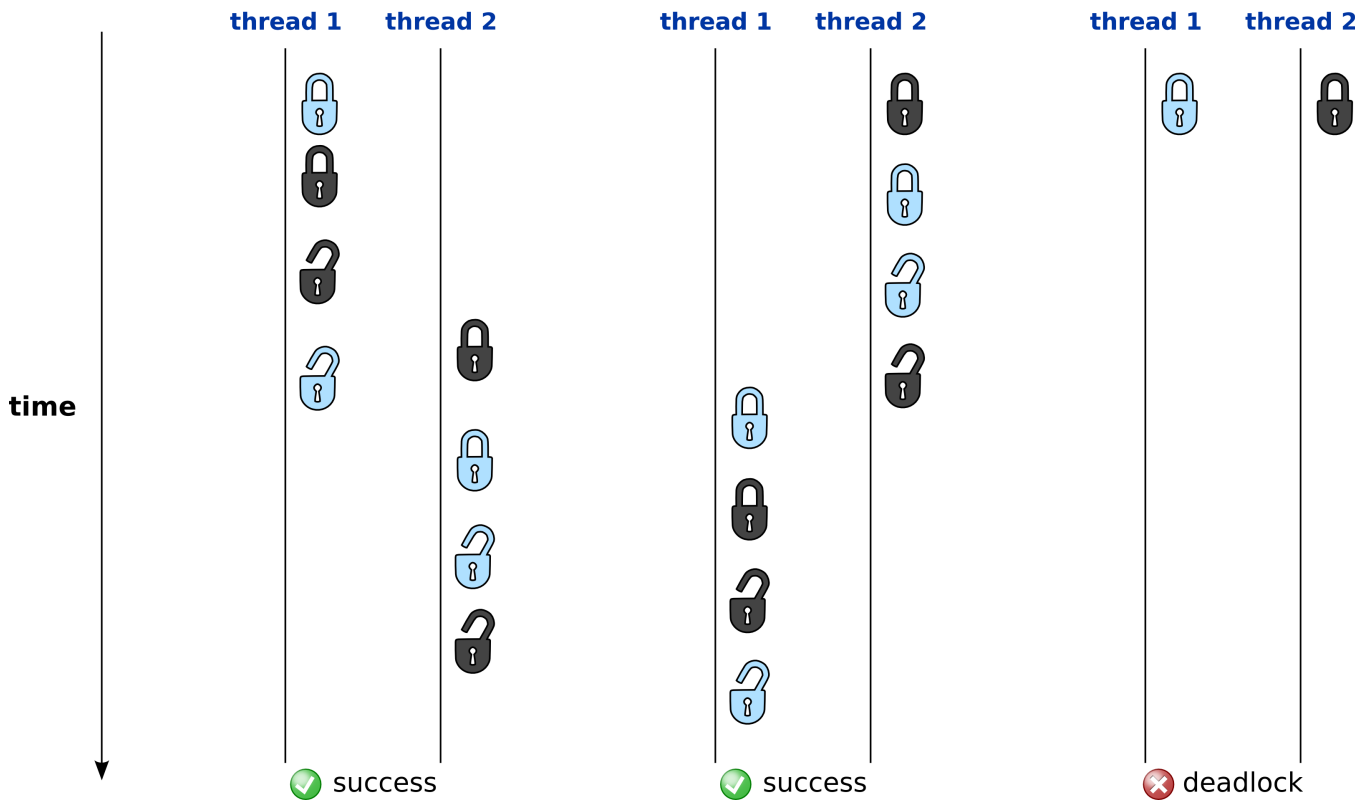
    error = pthread_mutex_unlock(&mutex_1);    // release mutex_1
    assert(error == 0);

    error = pthread_mutex_unlock(&mutex_2);    // release mutex_2
    assert(error == 0);

    return NULL;
}

```

Below is an example of three possible sequences of the lock/unlock events in the two threads.



Depending on timing the threads may run into a deadlock: the first thread has acquired mutex\_1 and is waiting for the mutex\_2, while the second thread has acquired mutex\_2 and is waiting for the mutex\_1.

Compile the code and run it several times:

```
$ gcc -g deadlock.c -o deadlock -lpthread
```

```
$ ./deadlock
```

If the deadlock occurs, the process blocks. Otherwise it runs to completion.

Now test `deadlock` with `maze`, running it 100 times<sup>15</sup>:

```
$ maze -r 100 ./deadlock > my_log16
```

Most of the runs complete successfully, yet some of them reveal the deadlock:

```
maze:ERROR: A deadlock was detected in the run # 59.  
maze:ERROR: A deadlock was detected in the run # 88.  
maze:ERROR: A deadlock was detected in the run # 100.
```

The stacks<sup>17</sup> of the threads in a deadlock can be found under run # 59, 88 or 100 in `my_log`:

```
maze:Run # 59  
.....  
maze:  
maze:The following processes are blocking:  
maze:  
maze:Main thread (tgid = 16029, pid = 16029) is waiting for a mutex.  
maze:  
maze:#0 0x55555425 in __kernel_vsyscall ()  
maze:#1 0x55578839 in __lll_lock_wait () from /lib/libpthread-2.8.so  
maze:#2 0x8048653 in main () at deadlock.c:30  
maze:  
maze:Thread (tgid = 16029, pid = 16030) is waiting for a mutex.  
maze:  
maze:#0 0x55555425 in __kernel_vsyscall ()  
maze:#1 0x55578839 in __lll_lock_wait () from /lib/libpthread-2.8.so  
maze:#2 0x804874b in simple_thread (dummy = 0) at deadlock.c:48  
maze:#3 0x5557232f in start_thread () from /lib/libpthread-2.8.so  
maze:#4 0x40820e in __clone () from /lib/libc-2.8.so  
maze:  
maze:Run # 59 completed with 1 error.
```

The deadlock can be reproduced by `maze` in the RECONSTRUCTION mode:

```
$ maze -R 59
```

---

<sup>15</sup> If `maze` detects no deadlocks during the TEST session (no error messages), try different *seed* values (“-s *seed*”), or increase the number of runs.

<sup>16</sup> Forwarding log to the file `my_log`; error messages will be printed to the terminal.

<sup>17</sup> The stacks may vary depending on the build type and the versions of `libc` and `libpthread`.