Threads

CISC 3350 — CUNY Brooklyn College
Lecture Notes

Threads

Threading is the creation and management of multiple units of execution within a single process.

- **Threads** are essentially lightweight processes that the current process creates in order to assign some work/tasks for these threads to complete.
- It is a major source of programming errors, however, due to data races and deadlocks, about which we will learn in these slides.

This chapter looks at the basics of the Linux threading API and will answer:

- How can threads be created in the C language?
- Why/when is using threads recommended?
- Data races and deadlocks - what are they, and how to prevent them?

Threads

Recall from the previous topic that **processes** are programs that are being executed by being loaded to memory.

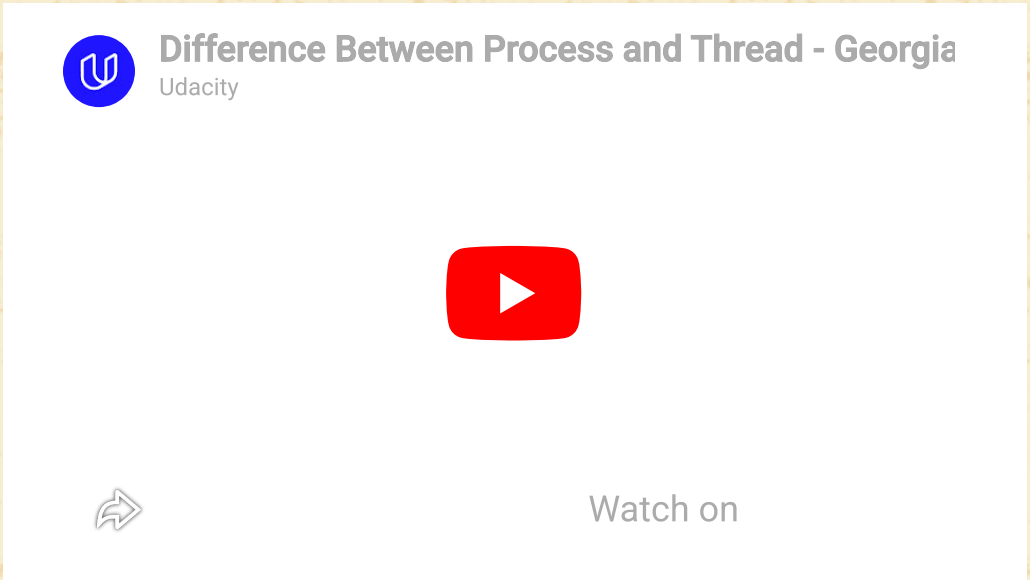
To increase the efficiency of running a program that contains similar tasks (like those found in loops) or independent tasks, operating systems support the execution of threads.

A **thread** is a basic unit of CPU execution, consisting of a thread ID, a program counter, a stack, and a set of registers. In the past, most processes were single-threaded; that is, each process had exactly one unit of execution, but most contemporary OSs support **multi-threading**, which is the ability of the OS to run threads concurrently.

Threads that belong to the same process will share the same text, data, and heap memory sections and the same process resources, including access to the files that were opened by the process and devices used by the process. The stack memory section, however, isn't common to all threads, as each thread has its own stack as mentioned above.

Threads

The following video explains the idea behind the behavior of threads:



https://www.youtube.com/watch?v=O3EyzlZxx3g

Benefits of Using Multiple Threads

- **Programming abstraction**: dividing up work and assigning a separate thread to a subtask. Since threads are independent, each thread is aware only of its own job, without getting into the details of other threads' jobs.
- **Parallelism and utilization**: with multiple processors, threads can achieve true parallelism, as each thread is an independently schedulable entity.
- **Improving responsiveness**: one thread might be dedicated to responding to user input, thus always keeping user interaction responsive.
- **Blocking doesn't influence other threads**: while one thread could be blocking (waiting for I/O or interrupt handling,) other threads will continue running.
- **Faster context switches**: the OS only needs to back up the thread's program counter, stack, and set of registers to memory, but doesn't need to flush the cache back to memory.
- **Memory saving**: threads of the same process share the same memory space, unlike several processes, each of which needs allocation of its own memory space.

Context Switch

When the execution of one process's instructions on the CPU must pause due to an interrupt that occurred, the OS must save the current execution state of the process before it handles the interrupt because the OS needs to run the interrupt handler software on the CPU.

In such a **state-save** action, the operating system copies the process's data from the CPU to memory. When the OS is ready to resume the execution of the process on the CPU, it performs a **state-restore** action, in which the process's data is copied from memory back to the CPU.

The same also happens when we remove one process from the CPU in favor of another process that will run there (remember that the OS does this often to create the sense that processes run simultaneously.) The procedure of saving the state of one process, removing it from the CPU, and then restoring the state of a different process on the CPU is called a **context switch**. Context switches happen extremely frequently, so the operating system performs them very fast.

Context Switch

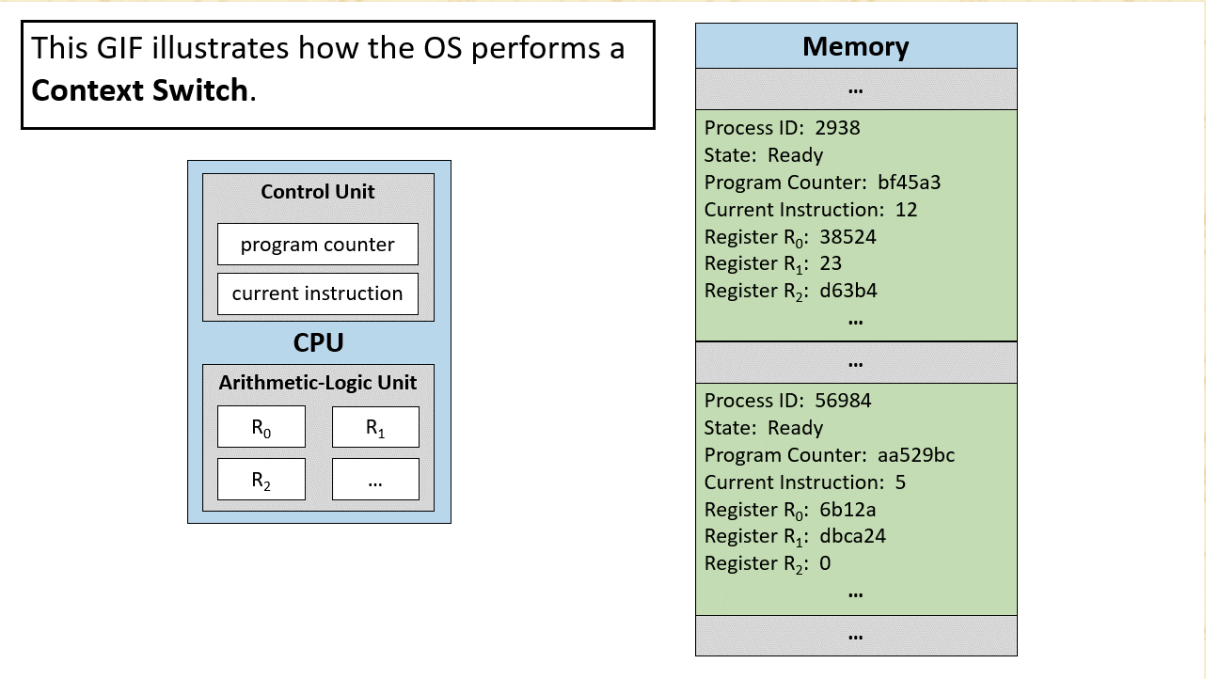


Diagram of a context switch. Miriam Briskman, CC BY-NC 4.0.

Note: click the GIF amination above to open and play it in a browser window.

Context Switching: Processes vs. Threads

Intra-process switching (switching between threads of the same process) is much faster than **inter-process switching** (switching between regular processes.) In Linux, the latter is near zero time!

Besides copying CPU registers to memory, process-switching also involves the somewhat time-consuming copy of data from cache to memory, since every process keeps its own data in the cache device.

Thread-switching doesn't involve copying the cache since threads share the same memory space, including the same data inside cache!

Challenges / Possible Issues of Multithreading

It is a challenge to identify:

- **Identifying Tasks**: What code sections in a program could be divided into tasks that could run concurrently.
- **Balance**: What code sections in a program could be divided into tasks that are roughly the same number of instructions so as not to waste CPU time when one thread runs for much more time than others.
- **Data Splitting**: What code sections in a program could be divided into independent tasks to be run by threads.
- **Data Dependency**: How to synchronize the execution of the threads. If the execution of thread B depends on the results of the execution of thread A, then running B before running A will cause some error in B's results or corrupt data.
- **Testing and debugging**: Whether the program runs correctly since timing and order of the execution of multiple threads can be unpredictable.

Issues of Multithreading

Possible alternatives to multithreading:

- Use multiplexed I/O or nonblocking I/O.
- Use *N* processes instead of *N* threads (which comes at the increased cost of context switching overhead.)
- Share memory between processes to reduce the cost of running multiple processes.

Still, threads are common both in system and other programming practices!

We will learn how to eliminate most of the dangers listed above at the end of this topic's lecture notes, when we learn how to synchronize threads.

Multithreading Models

We can identify two types of threads in contemporary operating systems: user threads and kernel threads.

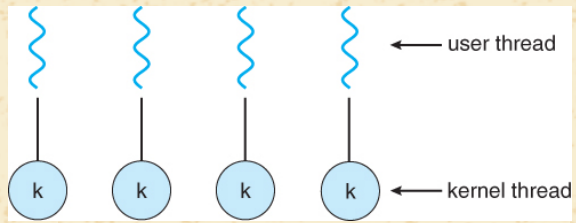
User threads are those coded by programmers and are used in user programs. Example: a multi-threaded Java program consists of user threads.

Kernel threads are internal threads that the operating system decides to create for simultaneous, and therefore efficient, execution of OS code.

Since user programs will resort to kernel services, we need to figure out how user threads will eventually be mapped to kernel threads. We describe three mapping models.

Multithreading Models

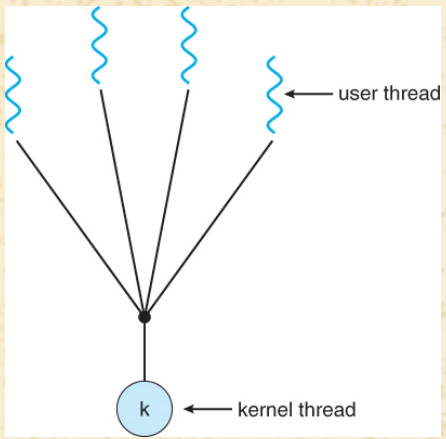
1. **One-To-One (1:1) Model**: For every user thread, the OS creates a separate kernel thread to handle it.
   - Also known as **kernel-level threading** since the kernel implements these threads as processes.
   - An API library creates new threads via the `clone()` system call, similar to `fork()`, which creates processes.
   - Since many kernel threads could be created (depending on the number of user threads,) the OS will spend more time switching between them, thus making the OS work slower.
   - What's good about this model is that, when one of the user threads blocks [= waits for I/O or some other activity], the other can continue to operate (since the underlying other kernel threads aren't blocking.)



One-to-one multithreading model. Taken from Bell, John T. "Threads." University of Illinois, Chicago.

Multithreading Models

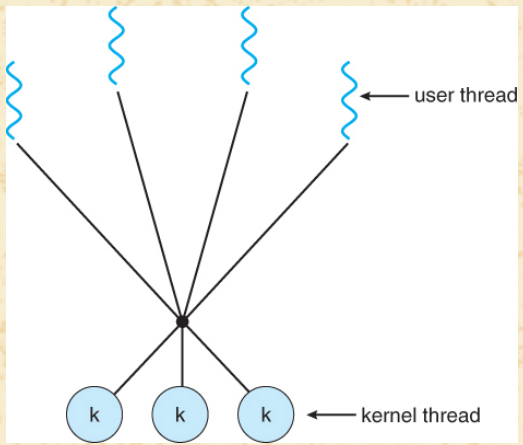
2. **Many-To-One (*N*:1) Model**: A process with *N* threads will map to a single kernel process.
   - It is also known as **user-level threading** since the implementation of the threads is user-program-dependent.
   - Thread management is handled by the thread library in the user space, which is very efficient.
   - The problem is that, when one of the user threads blocks, the other threads block as well since they all depend on a single kernel thread. In addition, due to the single kernel thread, which can run on only 1 CPU, threads can't run in true parallelism.



Many-to-one multithreading model. Taken from Bell, John T. "Threads." University of Illinois, Chicago.

Multithreading Models

3. **Many-To-Many (*N*:*M*) Model**: Several (*N*) user threads will map to several (*M*) kernel threads. Here, $N \geq M$.
   - Also known as ***N*:*M* threading** and as **hybrid threading**.
   - Attempts to achieve the best of both worlds (true parallelism from the 1:1 model, free context switches from the *N*:1 model.)
   - The number of created kernel threads will vary depending on the number of available CPUs, etc.
   - This model is difficult to implement, making 1:1 model the currently popular choice.



Many-to-Many multithreading model. Taken from Bell, John T. "Threads." University of Illinois, Chicago.

Multithreading Models

Any questions?

Some Computing Concepts

Parallelism (or **True Parallelism**) is when the instructions of two or more processes (or threads) run simultaneously on two or more CPUs (or multicore processors.)

Concurrence (or **Pseudo** [= false] **Parallelism**) is when the operating system uses context switches to quickly switch between processes (or threads) on a CPU to *create the feeling* that these processes (or threads) run in parallel, without them actually running in parallel.

Multi-programming is the execution of two or more programs at the same time by the OS. The idea is that the OS can store two or more programs in memory simultaneously while the programs wait for a CPU to become available.

Multi-processing is the use of two or more CPUs on the same device to run different parts of a process's code or to run two or more different processes. For example, process A could first run on CPU #0 and then on CPU #1 (or run two of its threads on both CPU #0 and CPU #1 in parallel.)

Some Computing Concepts

Multi-tasking is the ability of the OS to switch (fast) between processes (or threads) on a CPU to *create the feeling* that these processes (or threads) run in parallel, while they actually run concurrently. That is, multi-tasking is the ability of the OS to run processes (or threads) **concurrently**. [When there is more than one CPU, though, processes (or threads) could run **in parallel**.]

Multi-threading is the ability of the OS to switch (fast) between threads on a CPU to *create the feeling* that these threads run in parallel. That is, multi-tasking is the ability of the OS to run threads **concurrently**. [When there is more than one CPU, though, threads could run **in parallel**.]

Some Computing Concepts

Concurrent processes execute in an interleaving manner, "taking turns" on the same CPU, while parallel processes execute simultaneously on two or more CPUs:

- Illustration of Multitasking = Concurrency:

Process 1, CPU 1:

Process 2, CPU 1:

- Illustration of Parallelism:

Process 1, CPU 1:

Process 2, CPU 2:

Concurrency vs. Parallelism of processes. Miriam Briskman, CC BY-NC 4.0.

Note: click the GIF amination above to open and play it in a browser window.

Threading Patterns

Threading Patterns define the work behavior of the threads in your application.

There are two core programming patterns to choose from:

1. **Thread-per-connection**:
    - Each thread is assigned one unit of work for its entire execution duration.
    - The unit of work could be any group of activities you choose to assign.
        - For example, handling a client-server connection or some user I/O activity.
    - The total number of created threads depends on the OS or on your program.
        - Most OS have a limit (max) on the number of threads that can be created.
        - When this limit is reached, new requests/connections are either queued (waiting) or rejected.

Threading Patterns

1. **Thread-per-connection - Cont'**:
    - Issues with thread-per-connection threading:
        - This pattern may generate many threads.
        - The creation and maintenance of each thread have fixed time and space costs.
        - It creates scalability limits on the number of threads in each process, even when systems can handle thousands of connections.
        - Most threads in thread-per-connection are doing a lot of waiting: reading files, waiting for database servers to return results, etc.
            - This pattern is easier to code, but may not meet the needs due to the waiting.
    - Event-Driven Threading is the alternative to thread-per-connection (See next slide.)

Threading Patterns

2. **Event-Driven Threading**:
   - It spares threads from the need to wait for I/O.
     - Instead, it makes all the I/O activities asynchronous, and
     - Manages all I/O activities and requests via multiplexed I/O.
       - This procedure is called the **event loop**.
     - Only a ready I/O request needs the creation of a new thread.
   - No need to have more threads than CPU cores.
   - Currently, the preferred approach of the two is Event-Driven Threading.

Any questions?

Race Conditions

Two or more threads may race (= compete with) each other if they attempt to change data at the same moment.

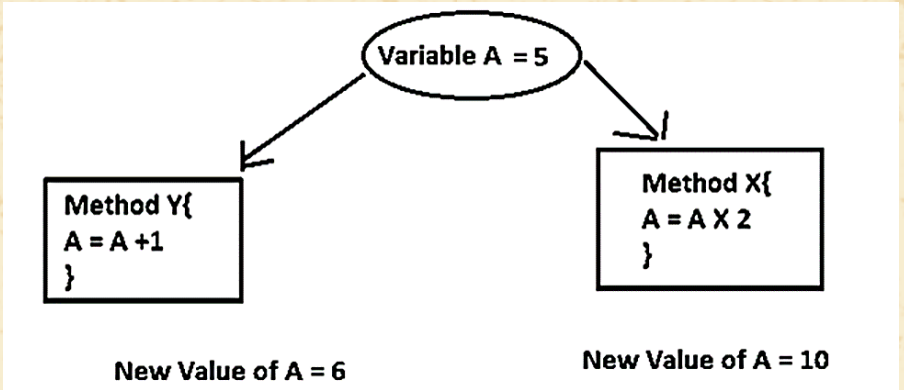
A **race condition** is a situation where the unsynchronized access of a shared resource by two or more threads leads to erroneous program behavior and corrupt data.

The shared resource could be anything the OS has to offer:

- Hardware (e.g., CPU, Memory, Disc Space, I/O devices, etc.)
- Data in memory: this is the most common form of a race, called a **data race**.

The part of the code where a race occurs is called a **critical region**.

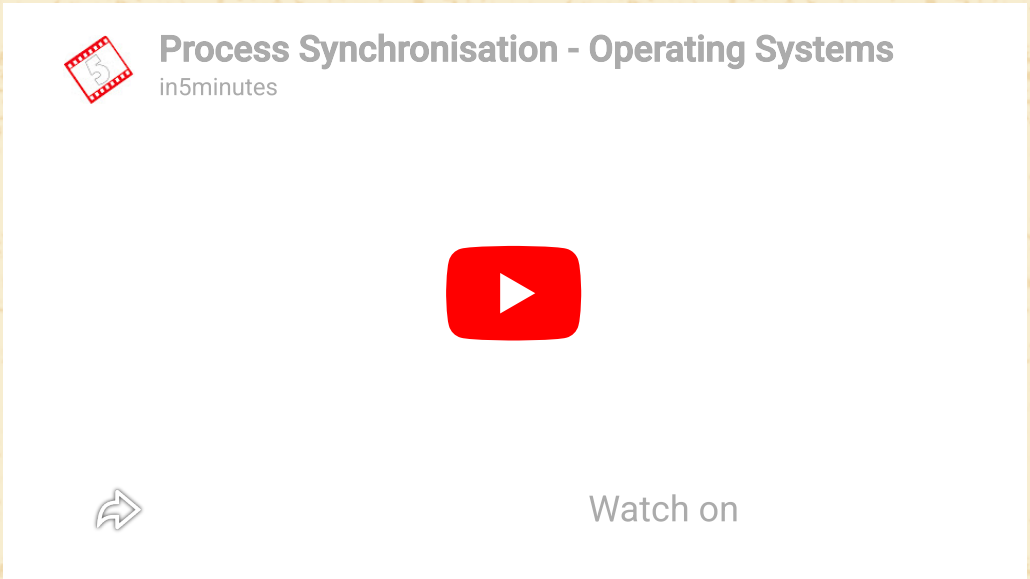
We can prevent races by **synchronizing** threads' access (= arranging an orderly and timed access) to critical regions.



Example of a race condition. Taken from this blog.

Race Conditions

This video introduces the **critical region** problem and presents a software solution to it:



https://www.youtube.com/watch?v=eKKc0d7kzww

Example of a Real-World Race: Withdrawing Money

Suppose a bank uses multi-threading for managing the withdrawal of money from customers' accounts (e.g., when a bill is scheduled to be paid.) If the order of the execution of the threads isn't controlled, serious race conditions will occur, and the result could be disastrous for the bank or the customers!

For example, the balance of the checking account of customer Jane Doe is now $500. She is withdrawing $200 via an ATM, and, at the same time, the bank is processing an electricity bill of $400. Here is what could go wrong:

- During the withdrawal of the $200, the thread sees that the balance is $500 (that is, there are enough funds to proceed,) so the withdrawal is processed. At the same time, the other thread that is trying to withdraw $400 sees that the balance is $500 (since it isn't aware of the withdrawal of the $200,) so this withdrawal is processed, too. This will result in a negative balance -$100 in Jane Doe's account!
- Another bad scenario is that, since the withdrawals happen at the same time, one of them might be cancelled out by the other one. That is, there is a chance that either the $400 or the $200 withdrawal will take effect, but not both, since one of the threads might overwrite the result of the other thread. So the result balance will be either $100 or $300!

Example of a Real-World Race: Withdrawing Money

The reason that simultaneous account balance changes could overwrite one another is that each of the code lines: `balance -= 200;` and `balance -= 400;` actually consists of 3 machine language instructions: (1) load the value of `balance` into a CPU register (e.g., R_1, R_2, ...) (2) add the requested sum, and (3) store the result back to main memory.

The order in which the 6 machine instructions run decides whether data overwriting (and, therefore, corruption) occurs. If the instructions don't mix (every 3 instructions execute separately), no overwriting occurs, and we get the balance of -$100:

Table 1: Sequential (Unmixed) Instructions

Time	Thread 1	Thread 2	R_1	R_2	balance
0	Load `balance` to R_1		500		$500
1	Add -200 to R_1		300		$500
2	Store R_1 to `balance`		300		$300
3		Load `balance` to R_2	300	300	$300
4		Add -400 to R_2	300	-100	$300
5		Store R_2 to `balance`	300	-100	-$100

or

Table 2: Sequential (Unmixed) Instructions

Time	Thread 1	Thread 2	R_1	R_2	balance
0		Load `balance` to R_2		500	$500
1		Add -400 to R_2		100	$500
2		Store R_2 to `balance`		100	$100
3	Load `balance` to R_1		100	100	$100
4	Add -200 to R_1		-100	100	$100
5	Store R_1 to `balance`		-100	100	-$100

Example of a Real-World Race: Withdrawing Money

However, here are scenarios in which machine instructions are mixed or execute in parallel: this overwrites `balance` and results in **corrupted data**. Note: we can't know in which order instructions will run, resulting in chaotic consequences!

Table 3: Interleaved (Mixed) Instructions

Time	Thread 1	Thread 2	R_1	R_2	balance
0	Load `balance` to R_1		500		$500
1	Add `-200` to R_1		300		$500
2		Load `balance` to R_2	300	500	$500
3	Store R_1 to `balance`		300	500	$300
4		Add `-400` to R_2	300	100	$300
5		Store R_2 to `balance`	300	100	$100

,

Table 4: Interleaved (Mixed) Instructions

Time	Thread 1	Thread 2	R_1	R_2	balance
0		Load `balance` to R_2		500	$500
1	Load `balance` to R_1		500	500	$500
2		Add `-400` to R_2	500	100	$500
3	Add `-200` to R_1		300	100	$500
4		Store R_2 to `balance`	300	100	$100
5	Store R_1 to `balance`		300	100	$300

,

Table 5: Parallel Instructions

Time	Thread 1	Thread 2	R_1	R_2	balance
0	Load `balance` to R_1	Load `balance` to R_2	500	500	$500
1	Add `-200` to R_1	Add `-400` to R_2	300	100	$500
2	Store R_1 to `balance`	Store R_2 to `balance`	300	100	$300 or $100

, etc.

unsynced_bank_withdraw.c: Race Condition during Bank Withdrawals


```c
/* This program demostrates the results of
 *    a race condition. While, in theory,
 *    we expect a specific number to be
 *    output, a different number may be
 *    generated in practice due to an
 *    existing race condition.
 *
 *    Miriam Briskman, 4/16/2023
 *    CISC 3350, Brooklyn College
 *    Licensed under CC BY-NC 4.0
 */

#include <stdio.h>
#include <stdlib.h>
```

unsynced_threads_experiment.c: Race Condition Experiment


```c
/* This program demostrates the results of
 *    a race condition. While, in theory,
 *    we expect a specific number to be
 *    output, a different number may be
 *    generated in practice due to an
 *    existing race condition.
 *
 *    Miriam Briskman, 4/16/2023
 *    CISC 3350, Brooklyn College
 *    Licensed under CC BY-NC 4.0
 */

#include <stdio.h>
#include <stdlib.h>
```

Race Condition Experiment

Any questions?

Synchronization

To prevent race conditions, the programmer needs to synchronize the execution of threads inside critical regions so that they are **mutually exclusive** (= only one thread runs a critical region at a time.)

An operation (or a set of operations) is **atomic** if it is indivisible and, hence, unable to be interleaved with other instructions.

To the rest of the system, an atomic operation appears to occur instantaneously: as a single instruction.

But that's the problem with computer instructions:

- They are not indivisible,
- They don't occur instantaneously, and thus:
- They aren't atomic.

How, then, could we prevent race conditions?

Mutexes

The most common technique for making critical regions atomic is by using a **lock**.

Because it ensures mutual exclusion, a lock is also known as a **mutex**.

The following real-life scenario is a good analogy to how a mutex works:

1. A lock guards a room so that only one person with the key can gain access to it.
2. There is only one key, which is initially placed on a table outside the room.
3. A person (a thread) comes to the door and opens it using the key.
4. Once inside, they lock the door from the inside so no one else may enter until that person leaves the room.
5. When the person decides to leave the room, they unlock the door and exit, leaving the key outside for the next person (another thread.)
6. If another person needs the room, go to step 3 above.
7. Repeat steps 3-6 until no more people (threads) need the room.

Mutexes: `lock()` and `unlock()` Functions

To force that only one thread at a time executes a piece of code, we will call a `lock()` function at its beginning, and an `unlock()` function at its end.

These two functions are provided with every library that manages threads; the C `pthreads` library, for example, calls these functions `pthread_mutex_lock()` and `pthread_mutex_unlock()`: we will discuss them in more detail in the 2nd half of this Topic.

In the following slides, the shown C programs call these functions to ensure that the code at the critical region(s) is accessed in mutual exclusive fashion, thus preventing race conditions!

Essentially, these programs include the 'fixes' to the programs on slides 28 and 29.

synced_bank_withdraw.c: Race Condition during Bank Withdrawals Fixed!


```c
/* This program demostrates the prevention
 *    of the a race condition present in
 *    unsynched_bank_withdraw.c
 *
 *    Miriam Briskman, 4/16/2023
 *    CISC 3350, Brooklyn College
 *    Licensed under CC BY-NC 4.0
 */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <errno.h>
```

synced_threads_experiment.c: Race Condition Experiment Fixed!


```c
/* This program demostrates that, with
 *    thread synchronization, no race
 *    conditions happen, so data isn't
 *    corrupted. This program includes
 *    a solution to the issue demonstrated
 *    in unsynced_threads_experiment.c
 *
 *    Miriam Briskman, 4/16/2023
 *    CISC 3350, Brooklyn College
 *    Licensed under CC BY-NC 4.0
 */

#include <stdio.h>
#include <stdlib.h>
```

Deadlocks

A **deadlock** occurs when two threads (or processes) are waiting for each other to release a resource and thus neither can complete its job since it waits forever.

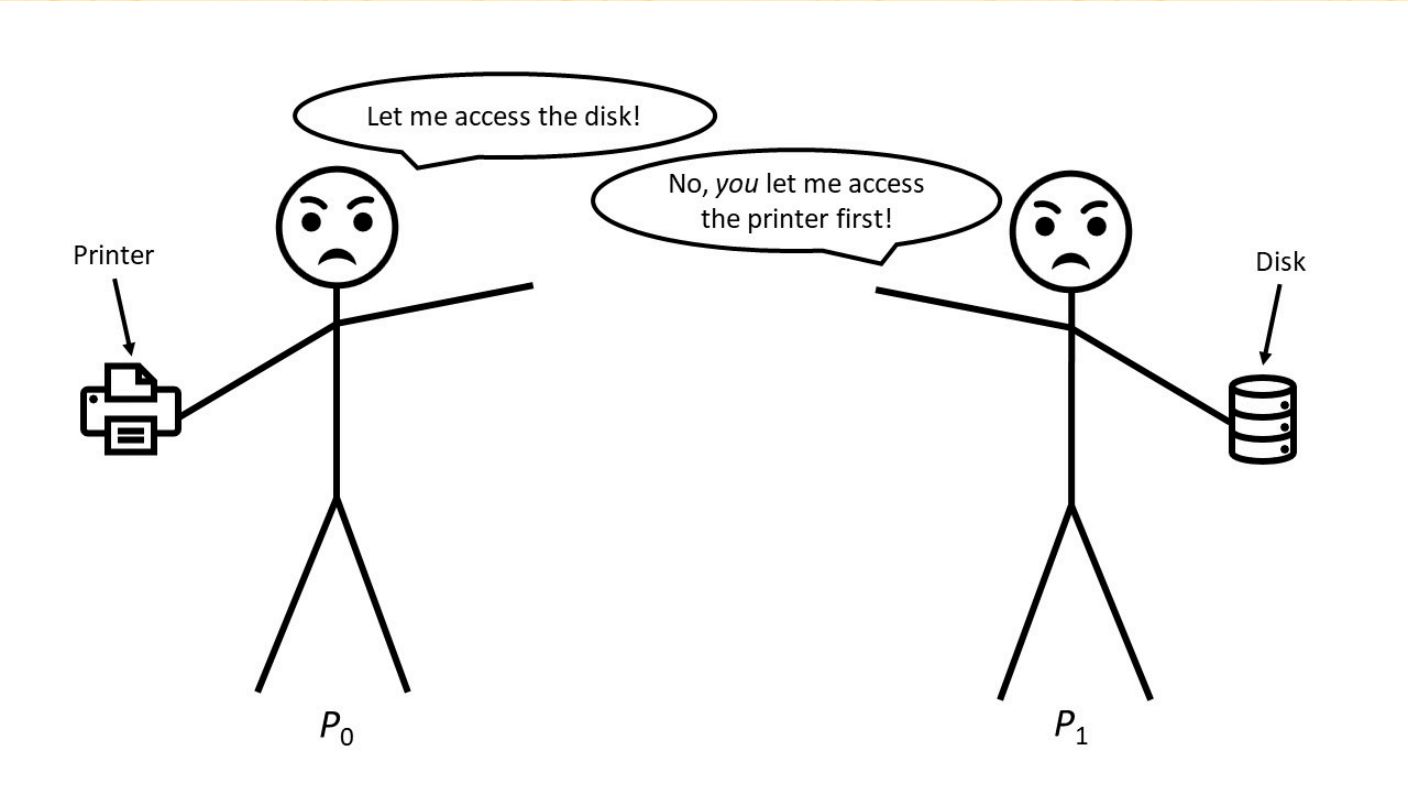
Two threads wait for a different mutex, which the other thread holds.

A **degenerate case** is when a single thread is blocked, waiting for a mutex that it already holds (like a dog chasing its own tail.)

How to Avoid Deadlocks:

- Have a clear hierarchy of data and understanding of what mutexes do to the data.
- Consider the ABBA deadlock. For example: Thread 1 holds resource A and Thread 2 holds resource B, and then when Thread 1 wants to use resource B and Thread 2 wants to use resource A, a deadlock happens.
- Fixing this situation requires clear rules: A thread that wants to use a resource must first free the resources that it holds.

Deadlocks



Example of a Deadlock on Two Processes. Miriam Briskman, CC BY-NC 4.0.

Deadlocks

Any questions?

Pthreads

Pthreads, or **POSIX Threads**, is Linux's C and C++ threading library.

Many large software projects define their own threading libraries:

- Examples: Android, Apache, GNOME, Firefox, etc.
- C++ and Java provide standard library support for threads.

POSIX standardized a threading library called "POSIX Threads", or Pthreads in short, for developers to use in their programs.

Pthreads remains the dominant threading solution for both C and C++ on Linux/Unix systems.

Pthreads

`Pthreads`, as a standard, is just an official document (not an implementation.)

In Linux, the implementation of this standard is provided by `glibc` (the GNU C Library), which provides 2 different implementations of `Pthreads`:

- **`LinuxThreads`** was Linux's original Pthreads implementation, but had unsatisfactory performance.
- **Native POSIX Thread Library** (**NPTL**) superseded LinuxThreads and became the standard Linux `Pthreads` implementation:
  - It features a 1:1 threading model based on the `clone()` system call and the approach that threads are implemented as lightweight processes that share resources, which is similar to LinuxThreads' approach.
  - NPTL makes use of new kernel interfaces, including the `futex()` syscall for synchronization, the `exit_group()` syscall for terminating all threads, and kernel support for thread-local-storage (TLS).
  - NPTL improves on LinuxThreads, allowing 1000s of threads in a single process without slowdown issues!

The Pthreads API

Provides over a 100 low-level interfaces to build a multithreaded program.

To use the library's function, `#include <pthread.h>` in your program's code.

Every `Pthreads` function, which BTW starts with `pthread_`, belongs to one of 2 categories:

- Thread management: functions to create, destroy, join (= wait for), and detach threads.
- Synchronization: functions to prevent race conditions; we cover only mutex lock functions in our course.

To make your program compile successfully, early `gcc` compiler versions (before `2.34`) require that you use the `-pthread` command-line flag when compiling a program that uses `Pthreads`. This is because `libpthread` is external to the `libc` standard library and thus requires explicit linking. Newer versions of `gcc`, however, don't require including `-pthread` during compilation.

```
$ gcc -Wall -Wextra -O2 -g -pthread -o program program.c
```

Creating Threads

<table>
<tr><td>Name: pthread_create()</td><td colspan="2">System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: creates a thread thread that runs the function start_routine, the only argument to which is arg.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <pthread.h> // Defines pthread_create().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);</code></td></tr>
<tr><td colspan="4">Description of arguments: thread: pointer to a pthread_t variable created before calling pthread_create().
attr: a pointer to a pthread_attr_t variable, which is used to change the default thread attributes of the newly created thread. Pass NULL for default attributes.
start_routine: the function that the thread will run. See more details on the next slide.
arg: the argument that will be passed to start_routine. It could be of any type (integer, char array pointer, NULL, etc.)</td></tr>
<tr><td rowspan="3" colspan="2">What type it returns: int</td><td colspan="2">On success: 0</td></tr>
<tr><td colspan="2">On failure: a non-zero error number that errno is usually set to.</td></tr>
<tr><td colspan="2">If failure, does it set errno? No</td></tr>
<tr><td colspan="4">Quick example: <code>if ((result = pthread_create (&thread, NULL, my_function, (void *) 543)) != 0) /* error */</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Creating Threads

When the `main()` function starts running, a single thread is created automatically: the **main thread**. We use `pthread_create()` to create additional threads.

The function whose name you pass to `pthread_create()` must be created by you as follows:

```
void * start_routine (void *arg);
```

Example:

```
void * factorial (void *num)
{
    long long fact = 1, i;
    for (i = 2; i <= (long long) num; i++)
        fact *= i;
    pthread_exit ((void *) fact); // We'll cover pthread_exit() soon!
}
```

Example of Creating a Thread


```
pthread_t thread1;
int result;
if ((result = pthread_create (&thread1, NULL, factorial, (void *) 10)) != 0)
{
    /* If 'result' is not a zero, we got an error! */
    errno = result; // We need this line for perror() to print the correct message:
    perror ("pthread_create");
    exit (EXIT_FAILURE);
}
```

A full C program is included at the end of these Topic's slides with another example of creating and waiting for threads!

Example of Creating a Thread

Any questions?

Thread ID

Name: `pthread_self()`	**System call or function:** Function	**Links:** Online Manual | Course Packet
What it does: returns a thread's ID. The returned type, `pthread_t`, is implementation-dependent, so it might not even be defined as an integer.		
What libraries you must include: `#include <pthread.h> // Defines pthread_self().`		
Syntax: `pthread_t pthread_self (void);`		
Description of arguments: – This function doesn't accept any arguments –		
What type it returns: `pthread_t`	**On success:** a data structure (maybe a struct) containing the ID of the thread.	
	On failure: – This function always succeeds: it never fails –	
	If failure, does it set `errno`? No	
Quick example: `printf ("thread ID = %d\n", pthread_self());`		
Other variations: N/A		

Thread ID

Name: `pthread_equal()`	**System call or function:** Function	**Links:** Online Manual | Course Packet
What it does: compares two thread IDs: returns a non-zero if equal, or 0 if unequal. Note again that **`pthread_t`** is not an integer, so you mustn't compare IDs by using the `==` operator!		
What libraries you must include: `#include <pthread.h> // Defines pthread_equal().`		
Syntax: `int pthread_equal (pthread_t t1, pthread_t t2);`		
Description of arguments: `t1`: the 1st compared thread ID `t2`: the 2nd compared thread ID		
What type it returns: `int`	**On success:** If the IDs are equal, a non-zero integer is returned. If they aren't equal, `0` is returned.	
	On failure: – This function always succeeds: it never fails –	
	If failure, does it set `errno`? No	
Quick example: `printf ("The IDs are %s.\n", (pthread_equal(t1, t2)) ? "equal" : "unequal");`		
Other variations: N/A		

Terminating Threads

Thread termination is similar to process termination, except that the rest of the threads in the process continue running.

A thread will terminate when either of the following happens:

- It returns from its `start_routine` (similar to a process returning from `main()`.)
- It calls the `pthread_exit()` function (similar to a process calling `exit()`.)
- It is canceled by another thread via `pthread_cancel()`.

The above events terminate a single thread.

All the threads of a process are terminated when the process itself terminates: (1) when the process returns from `main()`, (2) calls `exit()`, (3) or calls any of the `exec()` functions.

Terminating Threads

Name: `pthread_exit()`	**System call or function:** Function	**Links:** Online Manual | Course Packet
What it does: terminates the thread that calls this function, and returns the value at `retval` to the calling thread.		
What libraries you must include: `#include <pthread.h> // Defines pthread_exit().`		
Syntax: `void pthread_exit (void * retval);`		
Description of arguments: `retval`: the value that is returned. Pass `NULL` to return nothing.		
What type it returns: `void`	**On success:** – This function always succeeds: it doesn't return anything –	
	On failure: – This function always succeeds: it never fails –	
	If failure, does it set `errno`? No	
Quick example: `pthread_exit ((void *) -19.2);`		
Other variations: N/A		

In a few slides from now, we will see how the calling thread will retrieve the returned value: this is done by waiting for the thread to return.

How a Thread Can Terminating Other Threads

Name: `pthread_cancel()`	**System call or function:** Function	**Links:** Online Manual | Course Packet
What it does: terminates the thread `thread`.		
What libraries you must include: `#include <pthread.h> // Defines pthread_cancel().`		
Syntax: `int pthread_cancel (pthread_t thread);`		
Description of arguments: `thread`: the thread that we want to cancel		
What type it returns: `int`	**On success:** `0`	
	On failure: a non-zero error number that `errno` is usually set to.	
	If failure, does it set `errno`? No	
Quick example: `if ((result = pthread_cancel (thread)) != 0) /* error */`		
Other variations: N/A		

The termination occurs asynchronously: `pthread_cancel()` returns right away, and the thread exits when possible.

Whether and when a thread is cancellable depends on its **cancellation state** (enabled (default) or disabled) and **cancellation type** (asynchronous or deferred (default)), respectively.

Changing Cancellation State/Type

<table>
<tr><td>Name: <code>pthread_setcancelstate()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: changes the cancellation state of a thread.</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <pthread.h> // Defines pthread_setcancelstate().</code></td></tr>
<tr><td colspan="3">Syntax: <code>int pthread_setcancelstate (int state, int *oldstate);</code></td></tr>
<tr><td colspan="3">Description of arguments:
<code>state</code>: the state we want to set. Can be either <code>PTHREAD_CANCEL_ENABLE</code> (default state) or <code>PTHREAD_CANCEL_DISABLE</code>.
<code>oldstate</code>: a pointer to an integer. <code>pthread_setcancelstate()</code> will set this integer to the previous state.</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: a non-zero error number that <code>errno</code> is usually set to.</td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="3">Quick example:
<code>if ((result = pthread_setcancelstate (PTHREAD_CANCEL_ENABLE, NULL)) != 0) /* error */</code></td></tr>
<tr><td colspan="3">Other variations: N/A</td></tr>
</table>

Changing Cancellation State/Type

<table>
<tr><td colspan="2">Name: <code>pthread_setcanceltype()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: changes the cancellation type of a thread.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <pthread.h> // Defines pthread_setcanceltype().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int pthread_setcanceltype (int type, int *oldtype);</code></td></tr>
<tr><td colspan="4">Description of arguments:
<code>type</code>: the type we want to set. Can be either:
1. <code>PTHREAD_CANCEL_ASYNCHRONOUS</code>: thread may be killed any time – may leave process in an undefined state, or
2. <code>PTHREAD_CANCEL_DEFERRED</code>: thread will only be killed at specific, safe cancellation points, (default type).
<code>oldtype</code>: a pointer to an integer. <code>pthread_setcanceltype()</code> will set this integer to the previous type.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: a non-zero error number that <code>errno</code> is usually set to.</td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick example: <code>if ((result = pthread_setcanceltype (PTHREAD_CANCEL_ASYNCHRONOUS, NULL)) != 0) /* error */</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Example of One Thread Terminating Another


```c
// First, set the cancellation state and type
// (inside the to-be-terminated thread)
int old, res;
if ((res
     = pthread_setcancelstate (PTHREAD_CANCEL_ENABLE, &old)))
{
    errno = res;
    perror ("pthread_setcancelstate");
}
if ((res
     = pthread_setcanceltype (PTHREAD_CANCEL_DEFERRED, &old)))
{
    errno = res;
    perror ("pthread_setcanceltype");
}
```

```c
// Next, send the cancellation
request
// (from another thread)
int res;

// 'thread' is the thread to cancel:
if ((res = pthread_cancel (thread)))
{
    errno = res;
    perror ("pthread_setcanceltype");
}
```

Example of One Thread Terminating Another

Any questions?

Joining Threads

<table>
<tr><td colspan="2">Name: <code>pthread_join()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: joins (= waits for) a thread <code>thread</code> to terminate, and sets <code>retval</code> to the value that was returned by the thread via <code>pthread_exit()</code> or a <code>return</code> statement in the thread's function.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <pthread.h> // Defines pthread_join().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int pthread_join (pthread_t thread, void **retval);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>thread</code>: pointer to a <code>pthread_t</code> variable that was used for calling <code>pthread_create()</code>.
<code>retval</code>: a pointer to any variable or array. The <code>pthread_join()</code> function will set it to the value that the thread returned.
Pass <code>NULL</code> if you won't need to check or use the returned value.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: a non-zero error number that <code>errno</code> is usually set to.</td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick example: <code>if ((result = pthread_join (thread, &fact)) != 0) /* error */</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

After `pthread_join()` returns, we say that the 2 threads are **joined**. Joining allows threads to synchronize their execution.

Detaching Threads

<table>
<tr><td colspan="2">Name: <code>pthread_detach()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: detaches a thread <code>thread</code>: the thread is no longer joinable, so there is no need to join it. A detached thread automatically releases its resources upon exit from the thread's function.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <pthread.h> // Defines pthread_detach().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int pthread_detach (pthread_t thread);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>thread</code>: pointer to a <code>pthread_t</code> variable that was used for calling <code>pthread_create()</code>.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: a non-zero error number that <code>errno</code> is usually set to.</td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick example: <code>if ((result = pthread_detach (thread)) != 0) /* error */</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Rule of thumb: your program must call either `pthread_join()` or `pthread_detach()` (but not both) on every thread that you create with `pthread_create()`!

Otherwise, if a thread isn't joined nor detached, it will release it resources only after the process exits.

threads_example.c: Full C Program


```c
/* A C program using pthread_create() and
 *    pthread_join() to create 3 threads and manage
 *    their execution and termination.
 *
 *    Miriam Briskman, 4/16/2023
 *    CISC 3350, Brooklyn College
 *    Licensed under CC BY-NC 4.0
 */

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <errno.h>
```

Synchronizing Threads: Mutexes

Name: `pthread_mutex_lock()`	**System call or function:** Function	**Links:** Online Manual | Course Packet

What it does: acquires (= puts) a lock on a section of code: only one thread can access that code at any given time. In particular, this blocks the calling thread until `mutex` becomes available, which wakes up the calling thread, and it returns 0. If the `mutex` is available upon invocation, the function returns immediately.

What libraries you must include:
```
#include <pthread.h> // Defines pthread_mutex_lock().
```

Syntax:
```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

Description of arguments: `mutex`: pointer to a `pthread_mutex_t` variable that you must create and initialize (see example below) before calling `pthread_mutex_lock()`.

What type it returns: `int`	**On success:** `0`
	On failure: a non-zero error number that `errno` is usually set to.
	If failure, does it set `errno`? No

Quick example:
```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
if (pthread_mutex_lock (&mutex))
    /* Error */
```

Other variations: N/A

Synchronizing Threads: Mutexes

<table>
<tr><td colspan="2">Name: pthread_mutex_unlock()</td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: releases the lock mutex from a section of code.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <pthread.h> // Defines pthread_mutex_unlock().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int pthread_mutex_unlock (pthread_mutex_t *mutex);</code></td></tr>
<tr><td colspan="4">Description of arguments: mutex: pointer to a pthread_mutex_t variable that you used when you called pthread_mutex_lock().</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: int</td><td colspan="2">On success: 0</td></tr>
<tr><td colspan="2">On failure: a non-zero error number that errno is usually set to.</td></tr>
<tr><td colspan="2">If failure, does it set errno? No</td></tr>
<tr><td colspan="4">Quick example:

```
if (pthread_mutex_lock (&mutex)) /* Error */

    /* Some code lines: critical section where a race condition could happen */

if (pthread_mutex_unlock (&mutex)) /* Error */
```

</td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Synchronizing Threads: Mutexes

Any questions?

Threads: Sources

Topic 8: Threads. Briskman, Miriam. *CISC 3350 OER Course Packet.* 2024, CC BY-NC 4.0.
URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/CISC_3350_OER_Course_Packet.pdf#topic+8:+threads.

Partly based on the teaching of Professor **Yongqing Xiang** of Brooklyn College.