CITS3007 Secure Coding
Concurrency bugs and race conditions

Unit coordinator: Arran Stewart

Highlights

- Race conditions, data races, TOCTOU bugs
- Files and best practices
- Detecting and mitigating race conditions
- Data races in multithreaded programs

Introduction

Example problem

Suppose we have a setuid program, and we want to avoid giving the user access to a file unless they'd be able to access it normally.

Pseudocode:

```
res = real_uid_can_read_file("/tmp/somefile")
if not res:
  raise "user doesn't have access to /tmp/somefile"
# else access is OK
infile = open("/tmp/somefile", "r")
```

Example problem

Actual C code:

```
int res = access("/tmp/somefile", R_OK);
if (!res)
  abort();
int fd = open("/tmp/somefile", O_RDONLY);
```

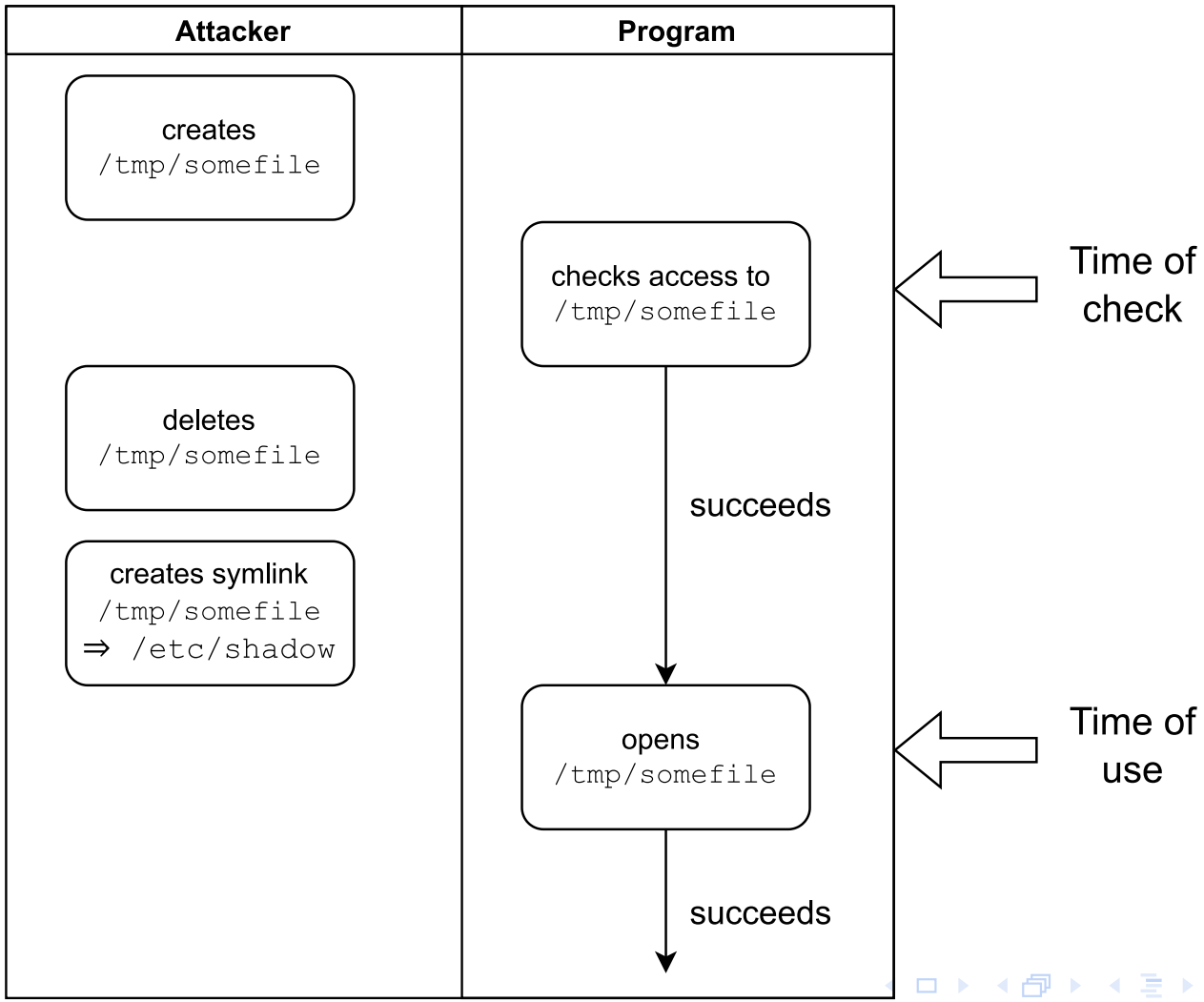
Example problem

```
int res = access("/tmp/somefile", R_OK);
if (!res)
  abort();
int fd = open("/tmp/somefile", O_RDONLY);
```

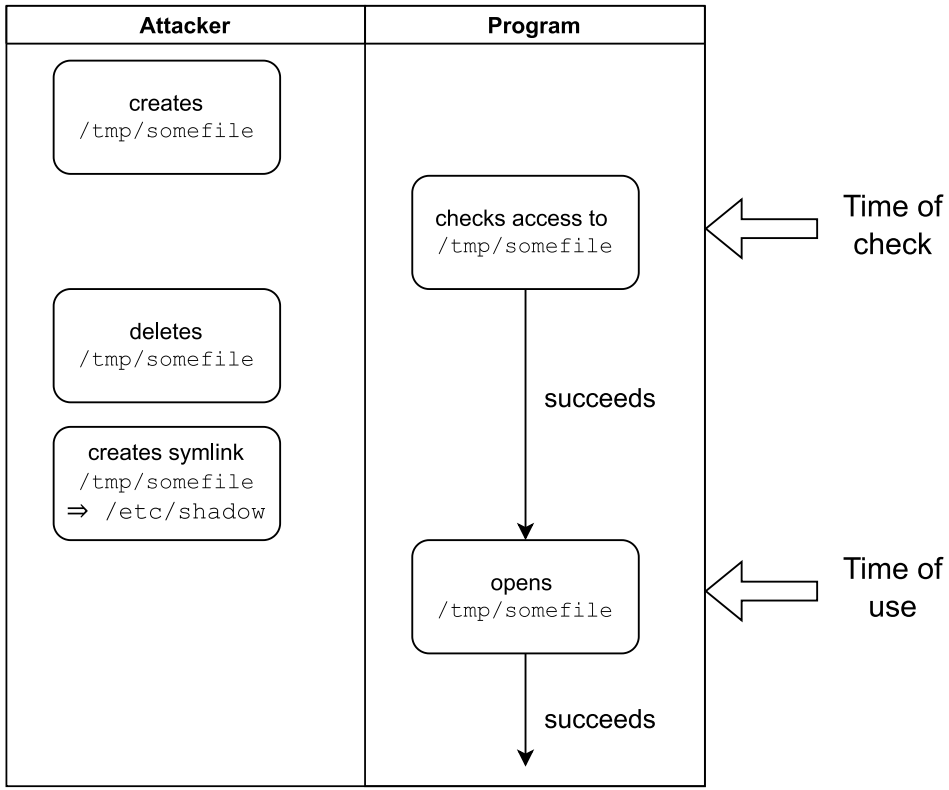
- Uses the `access(const char *pathname, int mode)` function
  - see "`man 2 access`" (man7.org)
- Checks `pathname` (dereferencing symbolic links) to see if the the calling process's *real* UID and GID are allowed to access the file
  - caller specifies whether to check read, write or execute access (or some combination)
- Answers question:
  "If I'm a setuid binary, can the user who invoked me read/write/execute this file?"

Example problem

Now suppose the following timeline of events occurs:



Example problem



This is called a “Time of Check to Time of Use” bug (TOCTTOU or TOCTOU).

How to exploit

Requires fairly precise timing. BUT an attacker

- may be able to just run attack repeatedly until it succeeds
- may be able to get a *notification* (see `man 7 inotify`) when file is created
- may be able to slow down the system, to give themselves more time
  - could overload the system with CPU-intensive tasks

Terminology

race condition when the timing or order of events affects the correctness of a piece of code, but the order of events is not controlled.

(Needn't involve threads or memory, specifically – the bug just described doesn't.)

data race in languages which permit concurrent access to memory: a situation where one thread is mutating some location in memory, and other threads access that same location and get or produce inconsistent or incorrect views of the same data.

Probably a subset of "race condition" (depending how exactly you define it).

TOCTOU bug a particular type of race condition where we *check* whether something's allowed, then *use* the results, but in between – whatever resource we were checking has been swapped with another

Data race example

data race

In languages which permit concurrent access to memory: a situation where one thread is mutating some location in memory, and other threads access that same location and get or produce inconsistent or incorrect views of the data.

Suppose some variable `mynum` is big enough that it requires two assembly instructions to write – e.g. a 64-bit `long long` on a 32-bit machine.

Thread **A** starts writing a value to `mynum`, but halfway through, thread **B** *reads* from `mynum`.

Thread **B** will get some junk value that's meaningless in the context of the program.

Problems

- Race conditions are always considered a *defect* or bug, because they make our program "flaky"
  - Our program does the right thing, **if** events happen in exactly the right order – but sometimes that won't be the case
- They *may* result in a vulnerability
- Historically, they have been a difficult type of bug to diagnose, because they can be hard to reproduce

Problems

TOCTOU vulnerabilities are a very common type of vulnerability due to a race condition, but there are plenty of others – here is one.

Suppose we have the following routine for performing a bank transfer (in Python-like pseudocode):

```
def perform_transfer(transfer_amount):
  balance = readBalanceFromDatabase()
  if transfer_amount < 0:
    raise Exception("Invalid transfer amount")
  new_balance = balance — transfer_amount
  if (balance — transfer_amount) < 0:
    raise Exception("Insufficient Funds")
  writeBalanceToDatabase(new_balance)
```

Problems

```
def perform_transfer(transfer_amount):
  balance = readBalanceFromDatabase()
  if transfer_amount < 0:
    raise Exception("Invalid transfer amount")
  new_balance = balance — transfer_amount
  if (balance — transfer_amount) < 0:
    raise Exception("Insufficient Funds")
  writeBalanceToDatabase(new_balance)
```

- The problem with this code is that it assumes the current transaction is the *only* one operating on a particular account
- A transfer is started by reading the balance from the database, and at the end, we write to the database – but in between, the current balance could have changed
- An attacker could use this timing issue to "give" themselves more money (can you suggest how?)

Transactions

```
def perform_transfer(transfer_amount):
  balance = readBalanceFromDatabase()
  if transfer_amount < 0:
    raise Exception("Invalid transfer amount")
  new_balance = balance — transfer_amount
  if (balance — transfer_amount) < 0:
    raise Exception("Insufficient Funds")
  writeBalanceToDatabase(new_balance)
```

- The problem arises because multiple threads of control are allowed to mutate a shared resource (the database).
- That's bad – the "low-level" solution is to *lock* the resource, so it can only be used by one thread at a time. Different languages offer different *synchronization primitives* for doing so (e.g. mutexes, semaphores)
- Databases typically offer a higher-level way of protecting against race conditions, *transactions*.

Transactions

```
def perform_transfer(transfer_amount):
  balance = readBalanceFromDatabase()
  if transfer_amount < 0:
    raise Exception("Invalid transfer amount")
  new_balance = balance — transfer_amount
  if (balance — transfer_amount) < 0:
    raise Exception("Insufficient Funds")
  writeBalanceToDatabase(new_balance)
```

- We mark a transaction using some sort of `start_transaction()` and `end_transaction()` procedure.
  - The semantics of transactions are: "A transaction is *atomic* – either the whole transaction occurs without error, or else the entire thing is rolled back and has no effect"
- Rather than locking the database, the DBMS "optimistically" allows multiple transactions to occur at once – but *if* it detects that they would interfere with each other, only one is allowed to proceed (the others are aborted)

Race conditions and file handling

Files and race conditions

- Files and directories are very common resources that can cause race conditions.
- Typically, multiple processes can open a file at once
  - though some OSs provide for enforced file-locking
  - Unix-like systems provide "advisory" (non-enforced) file locking via `fnctl()` (see `man 2 fnctl`)

Vulnerable to:

- Symbolic linking exploits
- Temporary file open exploits

Symlink exploits

The TOCTOU bug we saw at the start is a symlink exploit: an attacker alters some file by replacing it with a symbolic link.

Flakiness of filenames

Relying on a *file name* to always refer to the same file is *very* unreliable.

- Suppose we have `fileA` and `fileB`, and your program calls, say, `stat` (`man 2 stat`) to get some information about them (size or owner)
- But by the time your program accesses them, I've already run

  ```
  mv fileA tmp; mv fileB fileA; mv tmp fileA
  ```

- The files have been swapped

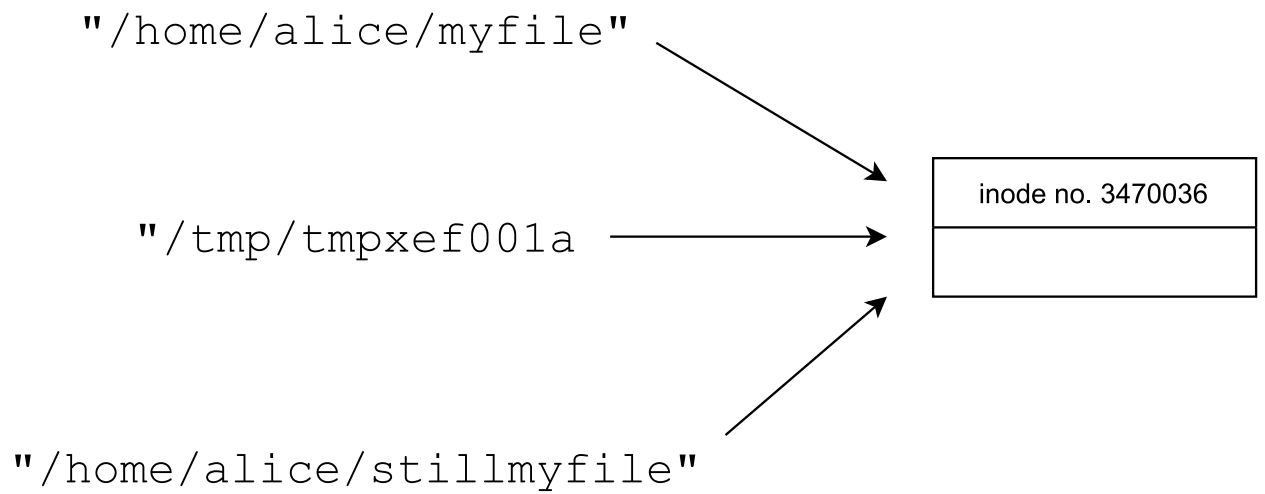
`stat` function

```
int stat(const char *pathname, struct stat *statbuf)
```

file paths vs inodes

Whenever a function refers to a file by *file path*, the kernel resolves the file path into something called the **inode** – a structure in the file system that uniquely describes some bunch of data on disk (like a file or directory).

Filenames might change (and multiple filenames might point to a single inode) – but once we've resolved a file path, opened the file and got an inode, we know it always refers to the same file.



file descriptors and inodes

And a *file descriptor* (as returned by `int open(const char *pathname, int flags)`) will refer to some specific inode.

So in general, if we want to ensure a file hasn't been "swapped out from under us", we should prefer functions that take file descriptors, rather than file names.

```
// prefer:
int fstat(int fd, struct stat *statbuf);
// over:
int stat(const char *pathname, struct stat *statbuf);
```

file-descriptor–based functions

For instance,

rather than ...	... prefer instead:
`chmod`	`fchmod`
`chown`	`fchown`
`unlink`	`unlinkat`
`rename`	`renameat`

Example of use – fchmod

```
// creates with default perms 0666 (-rw-rw-rw-)
FILE *fp = fopen("somefile", "w+")
if (!fp)
  abort();
// use f.d. to ensure we're always referring to same file
int fd = fileno(fp);
if (fchmod(fd, 0600) == -1)
  abort();
```

Temporary file bugs

```
char buf[1024];
strcpy(buf, "/tmp/tmpXXXX");
// use buf as template for a tempfile name
mktemp(buf)
if (buf[0] == '\0')
  abort();
// create file if it doesn't exist
// and open rw
int fd=open(buf, O_CREAT | O_RDWR, 0700);
```

- `mktemp()` (see `man 3 mktemp`) replaces the "`XXXX`" with random data, such that the new name does not exist
- If it couldn't generate a file that doesn't exist, `buf` is set to an empty string

Temporary file bugs

```
char buf[1024];
strcpy(buf, "/tmp/tmpXXXX");
// use buf as template for a tempfile name
mktemp(buf)
if (buf[0] == '\0')
  abort();
// create file if it doesn't exist
// and open rw
int fd=open(buf, O_CREAT | O_RDWR, 0700);
```

- `mktemp()` (see `man 3 mktemp`) replaces the "`XXXX`" with random data, such that the new name does not exist
- If it couldn't generate a file that doesn't exist, `buf` is set to an empty string
- Possible problems here?

Temporary file bugs

```
char buf[1024];
strcpy(buf, "/tmp/tmpXXXX");
mktemp(buf)
if (buf[0] == '\0')
  abort();
int fd=open(buf, O_CREAT | O_RDWR, 0700);
```

- The problems are bad enough that the man page for `mktemp` says "Never use mktemp()" (though not til you get to the "BUGS" section).

Problem 1:

- some implementations replace the X's with the process ID plus a single letter – very easy to guess
- Others use all random numbers for the X's
- But an attacker may be able to *force* creation of a particular filename, by using up all the other combinations
  (Especially if you don't have many X's. 1000 files is not at all many.)

Temporary file bugs

```
char buf[1024];
strcpy(buf, "/tmp/tmpXXXX");
mktemp(buf)
if (buf[0] == '\0')
  abort();
int fd=open(buf, O_CREAT | O_RDWR, 0700);
```

Problem 2:

- There's a race condition – we ask for a name, and then open it
- But in between, an attacker might create a file with the same name
- They might be able to inject malicious content into the file, or read confidential data from it

Recommended replacement:

- `int mkstemp(char *template);`
- Creates *and* opens a file, and gives us a file descriptor

Temporary file best practices

- Never reuse filenames, especially temporary files
- Use random file names for temporary files, using cryptographically strong random number generators
- Use `mkstemp()` instead of `mktemp()`, `tempnam()`, etc.
- Unlink (delete) your temporary files as early as possible
  - Reduces the window in which attacks can occur
- If possible, create your temporary files inside a temporary *directory* to which only you have access

Detecting and mitigating race conditions

Mitigating race conditions

Multiple approaches to mitigating a race condition:

- Remove concurrency
- Eliminate the shared resource
- Control access to the shared resource, so that it can't be unexpectedly changed

Mitigating race conditions

Remove concurrency

- Not always possible (e.g. we generally can't suddenly make an OS only have one process at a time)
- But for e.g. multithreaded programs: if we can remove threading completely, we make the program much easier to reason about
- If we can reduce the window of time in which races could occur, that also is an improvement
- We may be able to replace a *non*-atomic operation(s) (`mktemp()` and `open()`) with an *atomic* one (`mkstemp`)

Mitigating race conditions

Eliminate the shared resource

- Again, not always possible
- But: we should consider if we can reduce the number of shared resources we create.
- Can we use file *descriptors* instead of file names?
- Can we avoid use of shared directories (such as `/tmp`)?
- Could we e.g. use in-memory structures instead of files?
  - `mmap` (see `man 2 mmap`) will let us view a file as a `void*` buffer of memory
  - `memfd_create` (see `man 2 memfd_create`) creates an in-memory buffer, and lets us treat it like a file
  - With appropriate flags, mapped memory can be shared with child or sibling processes

Mitigating race conditions

Race conditions arise because the shared resource is *mutable* – multiple threads of control can change it in inconsistent ways.

Rather than get rid of it entirely, perhaps we can make it *immutable*.

Example:

- In Java, we can make collections immutable
  - e.g. to get an immutable `List` from an existing one, use
    ```
    List<Integer> immList = Collections.unmodifiableList(myList);
    ```
- It's now safe to access `immList` from multiple threads, since it's guaranteed never to change

Mitigating race conditions – locks

Control access to the shared object, so that it can't be unexpectedly changed

- We may be able to enforce a *lock* on the shared object
  - Hard to do this for files – but for e.g. variables in memory, many languages offer locks or `synchronized` sections.

Detecting race conditions

Often difficult to detect and reproduce.

Approaches:

- static analysis
- dynamic analysis

Detecting race conditions

`flawfinder` will detect some constructs that can lead to race conditions (see `man flawfinder`):

- CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ("Race Condition")

Detecting race conditions

For *data races* – one of the Google sanitizers is ThreadSanitizer (TSan)

- Helps detect data races.
- Typically slows program down by 5–10 times, uses 5–10 times more memory
- To use, compile and link with `-fsanitize=thread`
- By default, if a bug is detected, prints an error message to stderr.

Detecting race conditions

Fuzzing has been historically mostly applied to detecting memory errors, but can also be used for detecting concurrency errors. See:

- Jeong, Kim, Shivakumar, Lee & Shin, "Razzer: Finding Kernel Race Bugs through Fuzzing," 2019 IEEE Symposium on Security and Privacy (SP), doi: 10.1109/SP.2019.00017.

Data races

Data races

Data races occur when two or more threads access some shared variable

1. (potentially) at the same time, and
2. at least one of the accesses is a write.

(If all the accesses are reads – there's no chance for incorrect/inconsistent data to get created.)

Data races

Often result from assuming some (set of) operations are *atomic* when they're not.

- e.g. assuming a variable gets read/written in one CPU instruction

Java example

```
public class Number  {
  protected long number = 0;

  public void add(long value) {
    this.number = this.number + value;
  }
}
```

- The instruction

  `this.number = this.number + value`

  ins't guaranteed to be atomic.
- Other threads may get a view of `this.number` partway through updates

Data races

Safe multithreaded programming is a whole unit on its own.

But we look at general approaches.

Solutions – locking

In C, the "pthreads" (Posix threads) library is frequently used for multithreaded programs.

Provides functions for creating, locking and unlocking *mutexes* (mutually exclusive *locks*):

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
      const pthread_mutexattr_t *restrict attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

These are "*advisory* locks" – threads have to cooperate to use them properly, by acquiring a lock before accessing data.

Solutions – locking

For each bit of data you want to control access to: it's up to *you* to

- create a mutex that controls access to that data
- ensure that all code that uses the data acquires a lock on it first
- release the lock when finished (to increase concurrency, we generally lock the data for as short a time as possible)

Solutions – locking

- See `man pthreads` for an overview.
- Some C textbooks may also cover threading – but others may not (since it's a Posix standard, not part of the C language)
- Since C11, there's *also* a "native" C threading library, see `<threads.h>` on cppreference.com
  - Optional part of C11 – some compilers may not support it (GCC on Linux does)
  - Not well documented on many Linux systems – no `man` pages, on the CITS3007 development environment
- `pthreads` is somewhat more flexible and powerful.

Solutions – locking

This approach is easy to get wrong.

C mutex example

From SEI, "CON43-C. Do not allow data races in multithreaded code"

Code with race conditions:

```
static volatile int account_balance;

void debit(int amount) {
  account_balance -= amount;
}

void credit(int amount) {
  account_balance += amount;
}
```

The -= and += operations aren't atomic. Attacker can credit account with a large sum of money, and simultaneously make many concurrent withdrawals. Some withdrawals will probably fail to take effect because of the race condition.

C mutex example

Compliant solution using C11 mutexes:

```
#include <threads.h>
static int acct_bal;
static mtx_t acct_lock;

// returns -1 on error
int debit(int amount) {
 if (mtx_lock(&acct_lock) == thrd_error)
   return -1; // error
 acct_bal -= amount;
 if (mtx_unlock(&acct_lock) == thrd_error)
   return -1; // error
 return 0;    // success
}
```

```
// returns -1 on error
int credit(int amount) {
 if (mtx_lock(&acct_lock) == thrd_error)
   return -1;   // error
 acct_bal += amount;
 if (mtx_unlock(&acct_lock) == thrd_error)
   return -1;   // error
 return 0;      // success
}

int main(void) {
 if(mtx_init(&acct_lock, mtx_plain)
    == thrd_error
 )
 {
   /* Handle error */
 }
 /* ... */
}
```

Other options

Use the <stdatomic.h> header (introduced in C11)

```
#include <stdatomic.h>

atomic_int account_balance;

void debit(int amount) {
  account_balance -= amount;
}

void credit(int amount) {
  account_balance += amount;
}
```

Other languages

Other languages provide higher-level and more reliable constructs for dealing with locks.

```
class MyClass {
    private int sum = 0;

    public synchronized void calculate() {
        setSum(getSum() + 1);
    }

    // ... typical setters and getters
}
```

Java synchronized keyword

```
class MyClass {
    private int sum = 0;

    public synchronized void calculate() {
        setSum(getSum() + 1);
    }

    // ... typical setters and getters
}
```

e.g. In Java, a `synchronized` instance method causes Java to internally generate a lock for objects of type `MyClass`; any `synchronized` method automatically tries to acquire a lock when it starts, and releases it when done.

`synchronized` can also be used with blocks and static methods, but we don't cover that in this unit.

Synchronization in Python

Python does not have Java's nice "`synchronized`" keyword, so you have to write locks manually.

But the syntax is a bit more pleasant than in C (see here for details).