Concurrency and Security

PA193 Secure Coding Principles

Vladimír Štill
based on materials by Petr Ročkai

Faculty of Informatics, Masaryk University

Overview

1. Concurrent Programs
2. Race Conditions
3. Security Implications
4. Going Deeper (Hardware & More)
5. Valgrind

Examples mostly in C++(20) and C/POSIX API

Concurrent Programs

What is Concurrency?

- events that can happen at the same time
- it is not important if it *does*, only that it *can*
- events can be given a *happens-before* partial order
- they are *concurrent* if unordered by happens-before

```
std::atomic< int > x = 0, y = 0, z = 0;

void thread_a() {            void thread_b() {
    x = 1;                       if ( x > 0 ) {
    z = 1;                           y = 1;
                                     z = 2;
```

- `x = 1` and `x > 0` are concurrent
- `x = 1` happens before `y = 1`
- `z = 1` and `z = 2` are concurrent

Why Concurrency?

- *higher throughput* on multicore computers
- serving *multiple clients* at once
- *multiple tasks* that are largely independent

How?

- multiprocessing vs multithreading
  - different resource vs isolation trade-offs
- (some) asynchronous mechanism
- distributed computing over network

What is a Process?

- an *isolated* address space
- executing a single *program*
- owns OS-level resources
  - virtual memory
  - access to the CPU
  - open file descriptors
  - including network connections
- created by `fork()` on UNIX

Multiprocessing

- example: `httpd` (web server)
- each client connection gets a *new process*
- expensive: slow `fork`, needs more memory
- safe: *no interference* from other processes
- less safe but faster: process pools

What is a Thread?

- a sequence of instructions
- each physical CPU core can run 1 thread at a time
  - more with SMT-capable cores (2–8)
  - one process can contain many threads
- instructions within a thread run in a sequence
- no guarantees on operation ordering between threads
- also applies to threads from different processes

Multithreading

- think about `httpd` again
- each client connection gets a single thread
- threads are lightweight
- less context switching overhead
- further optimisation: thread pools

Multithreading

- think about `httpd` again
- each client connection gets a single thread
- threads are lightweight
- less context switching overhead
- further optimisation: thread pools

Multithreading in HPC

- HPC = high-performance computing
- threads can share data much more easily
- easier to write fast algorithms
- usually not security-relevant

The OS Kernel

- also runs concurrently with itself
- *many processes* can be doing system calls at once
- possibly preemptible
- “big kernel lock”: slows everything down
- preemptible kernels: fast but dangerous
    - Linux kernel is preemptible

Processes and Communication

- *IPC* = inter-process communication
- *message passing*: (relatively) safe but slow
- stdio, *sockets* or networks: even slower
- *shared memory*: fast but dangerous

Race Conditions

Shared Resources

- memory can be *shared* by multiple threads
- or even processes, through IPC mechanisms
- when is it *safe* to access/use a shared resource?

Shared Resources

- memory can be *shared* by multiple threads
- or even processes, through IPC mechanisms
- when is it *safe* to access/use a shared resource?

Critical Section

- any section of code that must *not be interrupted*
- the statement `x++` (`x = x + 1`) *could* be a critical section
- what is a critical section is domain-dependent
  - another example could be a bank transaction
  - or an insertion of an element into a linked list

Race Condition

- consider a *shared counter*, `i`
- and the following two threads

```
int i = 0;
void thread1() { i++; }
void thread2() { i--; }
```

What is the value of `i` after both finish?

Race Condition

- consider a *shared counter*, `i`
- and the following two threads

```
int i = 0;
void thread1() { i++; }
void thread2() { i--; }
```

What is the value of `i` after both finish?

Definition

- (anomalous) behaviour that *depends on timing*
- typically among *multiple threads* or processes
- an unexpected sequence of events happens
- recall that ordering is not guaranteed

Mutual Exclusion

- only one process (thread) can access a resource at once
- ensured by a *mutual exclusion device* (a.k.a *mutex*)
- a mutex has 2 operations: *lock* and *unlock*
- those must be correctly paired up
- *lock* may need to wait until another thread *unlocks*

```
int i = 0;
std::mutex m;
void thread1() { m.lock(); i++; m.unlock(); }
void thread2() { m.lock(); i--; m.unlock(); }
```

- POSIX: `pthread_mutex_{init,lock,unlock,destroy}`

Mutual Exclusion: Deadlocks

- happens if 2 or more threads *cannot proceed*
- each is *waiting* for a mutex locked by the *other thread*
- many other scenarios (not specific to mutexes)
  - files, sockets, other IPC, ...

```
std::mutex a, b;
void thread_1() { std::unque_lock lA( a );
                  std::unque_lock lB( b ); }
void thread_2() { std::unque_lock lB( b );
                  std::unque_lock lA( a ); }
```

Semaphore

- somewhat *more general* than a mutex
- allows *multiple* interchangeable *instances of a resource*
- and equal number of threads in the *critical section*
- basically an atomic counter

```
std::counting_semaphore< 2 > sem;
void t1() { sem.acquire();
            puts( "1 in" ); puts( "1 out" );
            sem.release(); }
void t2() { … }                     void t3() { … }
```

possible output:

`1 in; 3 in; 1 out; 2 in; 2 out; 3 out;`

- POSIX: `sem_init`, `sem_wait`, `sem_post`, `sem_destroy`

Monitors

- a *programming language* device (not OS-provided)
  - often encapsulated class
  - Java, Ada, ...
- internally uses standard mutual exclusion
- data of the monitor is only accessible to its methods
- only one thread can enter the monitor at once

Condition Variables

- what if a thread needs to *wait* for something?
- imagine a bounded queue implemented as a monitor
  - what happens if it becomes *full*?
  - the writer must be *suspended*
- condition variables have `wait` and `signal` operations
- connected with mutex or monitor

```
std::condition_variable cv; std::mutex m; int i = 0;

void t1() {
    std::unique_lock lk( m );
    cv.wait( lk, []{ return i == 1; } );
}
void t2() { i = 1; cv.notify_all(); }
```

- POSIX: `pthread_cond_*`

Spinlocks

- a *spinlock* is the simplest form of a mutex
- the `lock` method repeatedly tries to acquire the lock
  - this means it is taking up processor time
  - also known as *busy waiting*
- spinlocks contention on the same CPU is very bad
  - but can be very efficient *between* CPUs
  - waiting without extra context switches
- POSIX: `pthread_spin_*` (nothing in C++ standard)

Suspending Mutexes

- these need cooperation from the OS *scheduler*
- when lock acquisition fails, the thread *sleeps*
  - it is put on a *waiting queue* in the scheduler
- unlocking the mutex will *wake up* the waiting thread
- needs a system call $\rightarrow$ slow compared to a spinlock

Condition Variables Revisited

- same principle as a *suspending* mutex
- the waiting thread goes into a wait queue
- the `signal` method moves the thread back to a run queue
- many implementations can have spurious wakeup
- the busy-wait version is known as *polling*

Readers and Writers

- imagine a *shared database*
- many threads can read the database at once
- but if one is writing, no other can read nor write
- what if there are always some readers?

- C++: `std::shared_mutex`
- POSIX: `pthread_rwlock_*`

Shared Resources Revisited

- the *filesystem* is also a shared resource
- shared even between processes
- race conditions with other programs
  - possibly under the control of the attacker
- deadlocks without race conditions
  - writing to full `pipe`, ...
- same with *network resources* etc.

Security Implications

Two Types of Races

- within a *single application* (program)
  - bugs, not *necessarily* security-relevant
  - unexpected behaviour due to concurrency
  - eg. deadlocks/livelocks, memory corruption, etc.
  - races on file descriptors (`write` vs `close`)
- on *resources shared* with *third parties*
  - file system, network, etc.
  - almost always a security problem

Single-Program Races

- not always, but *sometimes security problems*
- CVE-2017-2636: race condition in the Linux kernel
  - unprivileged user can cause a timing-related double free
  - and possibly gain `root` privileges

The Systrace Race

- systrace was a BSD *syscall restriction* tool (sandbox)
- works by interposing every system call
- *inspected* at runtime by a *user-space program*
- syscall performed by the kernel if OK'd by the helper
- typical *check–perform* (TOC/TOU) race condition
  - replace argument between checking in wrapper and the actual syscall
  - path, other pointed-to arguments
  - file contents

Denial of Service: Deadlocks

- denial of service is a type of security problem
- the attacker can cause the system to malfunction
- deadlocks often lead to denial of service
- a deadlocked program cannot proceed executing

Non-Resource Deadlocks

- not all deadlocks are due to *resource contention*
- imagine a *message-passing* system
- process A is *waiting* for a message
- process B sends a message to A and *waits* for reply
- the message is *lost* in transit

File System: Permission Checks

- imagine a program is executing as `root`
- it can *send files* to users
- subject to standard *permission checks*
- what happens if it does `stat()` to check access
- then opens the file and sends content?

File System: Permission Checks

- imagine a program is executing as `root`
- it can *send files* to users
- subject to standard *permission checks*
- what happens if it does `stat()` to check access
- then opens the file and sends content?

Exploiting FS Races: Symlink Attacks

- the attacker creates, say, `/tmp/innocent`
- it *requests access* to that file via the above app
- *replaces the file* after the app does its `stat()`
- by a *symlink* pointing to, say, `/etc/shadow`

File System: Changing Ownership

- a program creates a file or a directory
- then calls `chown` to change the owner
- also *vulnerable* to symlink attacks
- CVE-2012-6095 (ProFTPd)

File System: Changing Permissions

- a file is written (with *sensitive content*)
- it's immediately `chmod`-ed
- but the *attacker can read* it in a narrow time window
- CVE-2013-2162
- solution:
  - set `umask` (for shell scripts)
  - pass restrictive `mode` to `open()`
  - `fchmod()` before `write()`

File System: Closing the Window

- file names are sensitive to symlink attacks
- but file *descriptors* are not
- `fchown()`, `fstat()`, `fchmod()` and so on
- *open first*, check using the file descriptor
- if the file is deleted, the fd still *points to original*

File System: Temporary Files

- race between picking a free name and creating a file
  - always use `O_CREAT`, `O_EXCL` for creation
  - never use `mktemp()`, use `mkstemp()` instead
- also applies to creating directories
  - never create with `mkdir -p`
  - either `mkdtemp()` or `mkdir`, `mktemp -d` with error checking
- should be created in a *safe location*
  - either owned by the same user as the process
  - or with the *sticky* permission bit set

Symlink Attacks: Not Just Races

- GDM did `chmod("/tmp/.X11-unix", 1777)`
- the attacker can *symlink* anything to `/tmp/.X11-unix`
- they get write access to that file
- instant `root` privileges
- CVE-2013-4169

Going Deeper (Hardware & More)

Modern CPU

- modern CPUs are very complex
  - up to 300 million transistors per core (2021)
  - pipelining, caches, speculative execution
  - relaxed memory behaviour
- many security features are rooted in hardware
  - process separation (privileged mode, virtual address space separation)
- separation must be enforced even if multiple processes are switched on the same CPU
  - danger of side-channel attacks
  - leaking of data through cache lookup speeds, speculation speeds ...

Meltdown & Spectre

- *hardware* vulnerabilities CVE-2017-5754, CVE-2017-5753, CVE-2017-5715, ...

Meltdown

- hardware race condition between instruction effect (including filling caches) and accesibility check
- reading from memory of other applications, kernel
  - passwords, keys, secret files, ...
- Linux workaround by stronger address-space separation between userspace and kernel $\rightarrow$ 5 % slowdown (with up to 30 % extremes)

Spectre

- class of branch-predictor-based vulnerabilities
- inside program memory space
- e.g. reading browser memory from JavaScript (cookies, passwords!)

SMT (Hyper-Threading)

- allows multiple threads to run on a single core
- this means such threads share certain resources
- this opens a window for side-channel attacks
- threads from different processes should not SMT
  - but in practice, this is often allowed

Speculative Execution, Pipelining

- instruction are not actually executed in-order of appearance
  - out-of-order execution, branch prediction
  - branches can be executes speculatively, later possibly invalidated
- pipelining and multi-issue execution means several instructions are executed at once
- effects are supposed to be hidden from single-threaded programs
  - but timing effects are visible
- can be very visible for *concurrent* programs

Atomic Variables, Memory Models

- low-level synchronisation can be done using atomic variables
- faster then locking if properly used
- much more error-prone

```
std::atomic< int > i = 0;
void thread1() { i++; }
void thread2() { i--; }
```

... result is *always* 0

- memory access between threads subject to memory model
  - delaying of writes (Intel & almost everyone)
  - delaying/reordering both reads and writes (ARM, POWER, ...)
  - recover ordering by atomics, synchronisation with mutexes, ...
  - any synchronisation is costly

Valgrind

Why Valgrind: Memory Safety

- we have seen many *memory bugs* so far
  - buffer overflows
  - use-after-free
  - double free
- C (and C++) are *memory unsafe*

Buffer Overflow

- *out-of-bounds* write to a buffer
- does not matter if heap or stack
- both are usually (and fatally) *exploitable*

Examples

- `gets()` … *never* use this function
- `scanf( "%s", buffer )` likewise
- `sprintf()`, `strcpy()`, etc. are often used wrong

Use After Free

- allocate some memory
- call `free` later, but retain the pointer
- read or (worse) write *through the pointer*
- usually *exploitable*

```
char *mem = malloc( 1024 );
if ( error )
    free( mem );
strncpy( mem, 1024, some_input );
```

Double Free

- call `free` on memory that was already freed
- usually causes *heap corruption*
- may very well be *exploitable*

```
char *mem = malloc( 1024 );
if ( error )
    free( mem );
// ...
free( mem )
```

Finding Memory Bugs

- memory bugs are notoriously *hard to debug*
- `valgrind` (specifically its `memcheck` tool)
- only finds bugs that were actually triggered by a test
- clean report does *not* mean your program is secure
- works by instrumenting/interpreting binary code

Helgrind

- *races* are *even harder* to find & fix than memory bugs
- use `valgrind` to detect concurrency issues (`helgrind` tool)
- *data races*, locking problems and so on
- you will learn more in the seminar

Some Other Tools

- *static*: LockLint (Sun)
  - fast but false positives
- *runtime*
  - address sanitiser, thread sanitiser (GCC/clang/MSVC)
  - Visual Threads (HP)
  - Thread Checker (Intel)
  - DRDT (Data Race Detection Tool; Sun)
- *verification*: DIVINE
  - slow but exact