

# Concurrency vs Security

Petr Ročkai

## Overview

- Part 1: Concurrent Programs
- Part 2: Race Conditions
- Part 3: Security Implications
- Part 4: Valgrind

# Part 1: 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**

## 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**

## 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`
- 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 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 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**

## Processes and Communication

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

```
void *thread( void *state )
{
    puts( "thread running" );
}

int main()
{
    pthread_t tid;
    pthread_create( &tid, NULL, thread, &x );
    puts( "main running" );
    pthread_join( tid, NULL );
}
```

## Example: C++

```
int main()
{
    auto f = [] { puts( "thread running" ); };
    std::thread t( f );
    puts( "main running" );
    t.join();
}
```

## Part 2: 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?



## Critical Section

- any section of code that must **not** be **interrupted**
- the statement  $x = x + 1$  **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: Example

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

```
int i = 0;
void thread1() { i = i + 1; }
void thread2() { i = i - 1; }
```

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

## Race Condition: 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**

## 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)

### Example

- 2 mutexes: A, B
- first thread locks A first, then B
- second thread locks B first, then A
- **race condition** on mutexes

## 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

## Monitors

- a programming **language** device (not OS-provided)
- 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 the monitor 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



## 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

## 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 → **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
- 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?

## Shared Resources Revisited

- the **file system** is also a shared resource
- shared even **between processes**
- race conditions with **other programs**
  - possibly under the control of the attacker
- same with **network resources** &c.

## Part 3: Security Implications

## Two Types of Races

- within a **single application** (program)
  - bugs, not **necessarily** security-relevant
  - **unexpected behaviour** due to sequencing
  - 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
- [a13xp0p0v.github.io/2017/03/24/CVE-2017-2636.html](https://a13xp0p0v.github.io/2017/03/24/CVE-2017-2636.html)



## 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

## 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 open the file and send 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()**

## 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` 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

## 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

# Part 4: 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
- **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**
  - Visual Threads (HP)
  - Thread Checker (Intel)
  - DRDT (Data Race Detection Tool; Sun)
- **verification**: DIVINE
  - slow but exact