How Uber Deceives the Authorities Worldwide

Uber has for years engaged in a worldwide program to deceive the authorities in markets where its low-cost ride-hailing service was resisted by law enforcement or, in some instances, had been banned.

The program, involving a tool called Greyball, uses data collected from the Uber app and other techniques to identify and circumvent officials who were trying to clamp down on the ride-hailing service. Greyball was part of a program called VTOS, short for “violation of terms of service,” which Uber created to root out people it thought were using or targeting its service improperly.

Administrivia

- Homework 5 due Thursday (3/9)
- Optional Section: VM problem + OS Q&A
- Course evaluation: https://uw.iasystem.org/survey/172382
- Final Exam: Tue, Mar. 14 @ 2:30pm in MGH 241
  - Review Session: Sun, Mar. 12 @ 1:30pm in SAV 264
  - Cumulative (midterm clobber policy applies)
  - TWO double-sided handwritten 8.5×11” cheat sheets
    - Recommended that you reuse/remake your midterm cheat sheet
Multiprocessing Revisited

- How are processes differentiated from one another?
- How are the processes’ execution contexts actually represented in memory?
What’s “In” a Process?

- A process consists of (at least):
  - An **address space** for the running program, containing
    - Code (instructions)
    - Data (static data, heap data, stack)
  - **CPU state**, consisting of
    - The program counter (%rip)
    - The stack pointer (%rsp)
    - Other register values (including PTBR, condition codes)
  - A set of **OS resources**
    - Open files, network connections, sound channels, ...

- Basically, all the stuff you need to run the program
  - Or to re-start it, if it’s interrupted at some point
The Process Namespace

- The name for a process is called a **process ID (PID)**
  - Just an integer

- The PID namespace is global to the system
  - Only one process at a time has a particular PID

- Operations that create processes return a PID
  - *e.g. fork()*

- Operations on processes take PIDs as an argument
  - *e.g. kill(), wait(), nice()*
    
    [be nice!]

Representation of Processes by the OS

- The OS maintains a data structure to keep track of a process’ state
  - Called the **process control block** (PCB) or process descriptor
  - Identified by the PID
- When the process isn’t running, the OS stores all of its execution state in (or linked from) the PCB
  - When *unscheduled* (moved to Ready or Blocked), execution state is transferred out of the hardware into the PCB
  - When Running, state is spread between the PCB and CPU
- Must be some fancy data structure that you’d never think of yourself, right?
The PCB

- The PCB is a data structure with many, many fields:
  - PID
  - Parent PID
  - Process state
  - Register values
  - Address space info
  - UNIX user id, group id
  - Scheduling priority
  - Accounting info
  - Pointers for state queues

- In Linux, defined in `struct task_struct`
  - [http://lxr.free-electrons.com/source/include/linux/sched.h](http://lxr.free-electrons.com/source/include/linux/sched.h)
  - Over 95 fields!
## Simplified View of a PCB

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process ID</td>
<td></td>
</tr>
<tr>
<td>Pointer to parent</td>
<td></td>
</tr>
<tr>
<td>List of children</td>
<td></td>
</tr>
<tr>
<td>Process state</td>
<td></td>
</tr>
<tr>
<td>Pointer to address space descriptor</td>
<td></td>
</tr>
<tr>
<td>Program counter</td>
<td></td>
</tr>
<tr>
<td>Stack pointer</td>
<td></td>
</tr>
<tr>
<td>(all) register values</td>
<td></td>
</tr>
<tr>
<td>uid (user id)</td>
<td></td>
</tr>
<tr>
<td>gid (group id)</td>
<td></td>
</tr>
<tr>
<td>euid (effective user id)</td>
<td></td>
</tr>
<tr>
<td>Open file list</td>
<td></td>
</tr>
<tr>
<td>Scheduling priority</td>
<td></td>
</tr>
<tr>
<td>Accounting info</td>
<td></td>
</tr>
<tr>
<td>Pointers for state queues</td>
<td></td>
</tr>
<tr>
<td>Exit (&quot;return&quot;) code value</td>
<td></td>
</tr>
</tbody>
</table>
Process States and State Transitions

- **Running**
  - dispatch / schedule
  - interrupt (unschedule)
  - terminate

- **Ready**
  - create
  - interrupt (I/O complete)

- **Blocked**
  - trap or exception (I/O, page fault, etc.)

- **Finished (“Zombie”)**
  - intentional
  - unintentional

You can create and destroy processes!
Process Execution States

- The process execution state indicates what it’s currently doing
  - Ready: waiting to be assigned to a CPU
    - Could run, but another process has the CPU
  - Running: executing on a CPU
    - The process that currently controls the CPU
  - Blocked (Waiting): waiting for an event
    - e.g. I/O completion, signal from (or completion of) another process
    - Cannot make progress until the event happens

- As a process executes, it moves from state to state
  - Linux: run `ps -x` and STAT column shows current state
State Queues

- The OS maintains a collection of queues, typically one for each state
  - Each PCB is found in the queue of the execution state its process is currently in
  - When a process changes state, its PCB is unlinked from one queue and linked onto another

- Once again, *this is just as straightforward as it sounds!* The PCBs are moved between queues, which are represented as linked lists. *There is no magic!*
State Queues

- Ready queue header:
  - head ptr
  - tail ptr
  - chrome (1365)
  - vim (948)
  - ls (1470)

- Wait queue header:
  - head ptr
  - tail ptr
  - cat (1468)
  - chrome (1207)

- There may be many wait queues, one for each type of wait (particular device, timer, signal, ...)

- These are PCBs
PCBs and State Queues

- PCBs are dynamically allocated inside OS memory
- When a process is created:
  1) OS allocates a PCB for it
  2) OS initializes PCB
  3) (OS does other things not related to the PCB)
  4) OS puts PCB on the correct queue
- As a process computes:
  - OS moves its PCB from queue to queue
- When a process is terminated:
  - PCB may be retained for a while (to receive signals, etc.)
  - Eventually, OS deallocates the PCB
The Big Picture

- Single processor can run many processes via scheduling
  - Concurrency!

- What if we have multiple processors?
  - Parallelism! (and concurrency)
  - Option 1: Run multiple processes simultaneously
  - Option 2: Run different parts of the same process simultaneously – how?
What’s Needed?

- Working on the same process, so each processor wants access to:
  - The same code/instructions
  - The same data (shared)
  - The same privileges
  - The same resources (open files, network connections, etc.)

- But need multiple hardware execution states:
  - An execution stack and stack pointer
  - The program counter
  - A set of general-purpose processor registers and their values
Can We Use Processes?

- Given the process abstraction as we know it:
  - Fork several processes
  - Share data by mapping each to the *same* physical memory
    - See the `shmget()` system call for one way to do this (kind of)

- This is really inefficient!
  - New PCB
  - New page table
  - New address space
  - Etc.
Introducing Threads

- Separate the concept of a process from that of a minimal “thread of control”
  - Usually called a thread (or a **lightweight process**), this is a sequential execution stream within a process

- In most modern OS’s:
  - **Process**: address space, OS resources/process attributes
  - **Thread**: stack, stack pointer, program counter, registers
  - Threads are the **unit of scheduling** and processes are their containers
Communicating Between Threads

- A thread is bound to a single process/address space
  - The address space provides isolation from other processes
  - Communicating between processes is expensive
    - Must go through the OS to move data from one address space to another

- A single process/address space can have multiple threads executing within it
  - Communicating within a process is simple/cheap
    - Just update a shared variable!
  - Creating threads is cheap, too
The Design Space

Key

<table>
<thead>
<tr>
<th>address space</th>
<th>thread</th>
</tr>
</thead>
</table>

| MS/DOS        |        | older UNIXes |
| one thread per process | one process | many processes |

| Java          |        | Mach, NT, Chorus, Linux, … |
| many threads per process | one process | many processes |
(OLD) Process Address Space

Address space:

- Code (text segment)
- Static Data (data segment)
- Heap (dynamic allocated mem)
- Stack (dynamic allocated mem)

Address space boundaries:

- 0x00000000
- 0xFFFFFFFF

Registers:

- %rip
- %rsp

Shared memory segment.
(NEW) Address Space with Threads

- **Thread 1 Stack**
- **Thread 2 Stack**
- **Thread 3 Stack**
- **Heap** (dynamic allocated mem)
- **Static Data** (data segment)
- **Code** (text segment)

Address space:
- 0x00000000
- 0xFFFFFFFF

- %rsp (T1)
- %rsp (T2)
- %rsp (T3)
- %rip (T2)
- %rip (T1)
- %rip (T3)

Individual threads usually don't use much stack mem.
Multithreading

- Multithreading *(separating processes and threads)* is useful for:
  - Handling concurrent events
  - Building parallel programs

- Supporting multithreading is a big win
  - Creating concurrency does not require creating new processes
  - Even useful on a uniprocessor
    - Only one thread can run at a time, but it is a useful program structuring mechanism
What’s “In” a Thread?

- The state of a thread is maintained in a data structure called the **thread control block (TCB)**
  - An identifying integer: the **thread ID (TID)**
  - Register values (\%rip, \%rsp, etc.)
  - Thread state (Running, Ready, Blocked, Finished)
  - Scheduling priority
  - Pointer to PCB (**shared state**)

- It is now the TCBs that are placed on the state queues instead of the PCBs
  - Managed by the OS’s **thread scheduler**
User-level Threads

- **Note**: there are *kernel* threads and *user* threads
  - We will only talk about user threads here

- Threads managed at the user level (that is, entirely from within the process)
  - A user-level library manages the threads
    - *e.g.* pthreads, OpenMP
  - Creating a thread, switching between threads, and synchronizing threads are done via procedure calls
  - Thread manager doesn’t need to manipulate address spaces
    - Threads differ (roughly) only in hardware contexts (registers), which can be manipulated by user-level code
Thread Context Switch

- Very simple for user-level threads:
  - Save context of currently running thread
    - Save CPU state in TCB (sometimes push onto thread stack)
  - Restore context of the next thread
    - Restore CPU state from next thread’s TCB (or pop from stack)
  - Return as the new thread
    - Execution resumes at program counter of next thread
  - No changes to memory mapping required!

- This is all done in assembly!
Multithreading Headaches

- Parallelizing code can be difficult
  - Possible speedup limited by amount of computation that can be parallelized (Amdahl’s Law)

- Cache coherence
  - If different CPUs are writing data in the same address space simultaneously, what happens to data consistency between caches and memory?
    - 1. Update caches for other CPUs simultaneously
    - 2. Invalidate block in other CPUs’ caches
    - “4th C of cache misses”: coherence

- Synchronization
  - Coordinate events to avoid non-deterministic behavior
  - Often arises with writing to shared data or executing a critical section of code
**Multicore Multithreading Issues?**

**Processor package**

Core 0
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache

Core 3
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache

L3 unified cache (shared by all cores)

**Main memory**

**Block size:**
- 64 bytes for all caches

**L1 i-cache and d-cache:**
- 32 KiB, 8-way,
  - Access: 4 cycles

**L2 unified cache:**
- 256 KiB, 8-way,
  - Access: 11 cycles

**L3 unified cache:**
- 8 MiB, 16-way,
  - Access: 30-40 cycles
Topics

- **Secondary Storage – Disks**
- **File Systems**
  - Files, directories, and disk blocks
An Example Memory Hierarchy

- **registers**: <1 ns
- **on-chip L1 cache (SRAM)**: 1 ns
- **off-chip L2 cache (SRAM)**: 5-10 ns
- **main memory (DRAM)**: 1-2 min
- **local secondary storage (local disks)**: 15-30 min
- **remote secondary storage (distributed file systems, web servers)**: 31 days

**Storage Characteristics**

- Smaller, faster, costlier per byte
- Larger, slower, cheaper per byte

**Performance Latency**

- SSD: 1-150 ms
- Disk: 1-150 ms
- Remote secondary storage: 1-15 years
Secondary storage

- Secondary storage is anything outside of “primary memory”
  - Does not permit direct execution of instructions or data retrieval via machine load/store instructions

- Characteristics:
  - **Large**: 512 GB – 2 TB+ (and growing!)
  - **Cheap**: < $0.05/GB for hard drives
  - **Persistent**: data survives power loss
  - **Slow**: milliseconds to access
  - **Not immune to failure**: device failures and read/write errors, though rare
Disk Capacity Trends

- **1975-1989**
  - 25% improvement/year → 2x/37 months, 9.3x/decade
  - Exponential, but slower than processor performance

- **1990-recently**
  - 73% improvement/year → 2x/15 months, 240x/decade
  - Capacity growth faster than processor performance


https://www.forbes.com/sites/tomcoughlin/2012/10/03/have-hard-disk-drives-peaked/
Storage Latency: How Far Away is the Data?

- **Tape/Optical Robot**: 2,000 Years
- **Disk**: 2 Years
- **Memory**: 1.5 hr
- **On Board Cache**: 10 min
- **On Chip Cache**: 1 min
- **Registers**: 1 min

Latency Times:
- **Tape/Optical Robot**: 10^9
- **Disk**: 10^6
- **Memory**: 100
- **On Board Cache**: 10
- **On Chip Cache**: 2
- **Registers**: 1

© 2004 Jim Gray, Microsoft Corporation
Physical Disk Structure

- Disk components
  - platters
  - surfaces
  - tracks
  - sectors
  - cylinders
  - arm
  - heads

7200 rpm $\leftrightarrow$ 120 rps (slow!)
Interacting with Disks

- In the old days, OS would have to specify cylinder #, sector #, surface #, transfer size
  - *i.e.* OS needed to know all of the disk parameters

- Modern disks provide a higher-level interface (e.g. SCSI)
  - Maps **logical blocks** to cylinder/surface/sector
  - OS and disk interface via logical block #’s
  - Disk controller, on-board cache (**disk buffer**)
  - Physical parameters are hidden from OS – both good and bad
Disk Performance Optimizations

- **Disk Layout**
  - OS may increase file block size in order to reduce seeking
  - OS may seek to co-locate “related” items in order to reduce seeking (defragmentation)

- **Caching**
  - Keep data in memory to reduce physical disk access

- **Pre-fetching**
  - If file access is sequential, fetch blocks into memory before requested (spatial locality!)

- **Scheduling of Disk Requests**
  - Different algorithms to address request queue
Solid State Drives

- Hard drives are based on spinning magnetic platters
  - Performance determined by the *mechanics* of the drives
  - Cost dictated by massive economies of scale, and many decades of commercial development and optimization

- Solid state drives are based on NAND flash memory
  - No moving parts – more like RAM than spinning disk
  - Performance driven by *electronics* and *physics*
  - Price is still quite high in comparison to hard drives, but dropping fast

SSD Performance

- **Reads**
  - Unit of read is a *page*
  - 0.01 – 0.1 ms read latency (50-1000x better than disk seeks)
  - 40-400 MB/s read throughput (1-3x better than disk throughput)

- **Writes – slower than reading!**
  - Unit of write is a *page*
  - Flash media must be *erased* before it can be written to
    - Unit of erase is a block, typically 64-256 pages long
    - Blocks can only be erased a certain number of times before they become *unusable!* – typically 10,000-1,000,000 times
SSD Performance Optimizations

- Virtualize pages and blocks on the drive
  - *i.e.* Expose logical pages, not physical pages, to the rest of the computer

- Wear-leveling
  - When writing, try to spread erases out evenly across physical blocks of the SSD
  - Intel promises 100GB/day x 5 years for its SSD drives

- Log-structured filesystems
  - Convert random writes within a filesystem to log appends on the SSD

- Build drives out of arrays of SSDs, add lots of cache
Disks and the OS

- Disks are messy, messy devices
  - errors, bad blocks, missed seeks, etc.

- Job of OS is to hide this mess from higher-level software
  - Low-level device drivers (initiate a disk read, etc.)
  - Higher-level abstractions (files, databases, etc.)
  - Common reality: disk has disk buffer, OS uses page cache

- OS may provide different levels of disk access to different clients