Multiprocessing (part 2)

Ryan Eberhardt and Armin Namavari

April 30, 2020

Project logistics

- Project (mini gdb) coming out tomorrow, due May 18
- You’re also welcome to propose your own project! Run your idea by us before you start working on it
    - Rust tooling (e.g. annotate code showing where values get dropped)
    - Write a raytracer
    - Pick a command-line tool and try to beat its performance (e.g. grep)
    - Implement a simple database

Today

- (From last time) Why you shouldn’t use signal() 🔥🚒
- Multiprocessing case study of Google Chrome

Don’t call signal()

signal() is dead. Long live sigaction()

signal(2) - Linux man page

Name

signal - ANSI C signal handling

Synopsis

#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int *signum*, **sighandler_t** *handler***);**

Description

The behavior of **signal**() varies across UNIX versions, and has also varied historically across different versions of Linux. **Avoid its use**: use sigaction(2) instead. See *Portability* below.

signal() is dead. Long live sigaction()

> **Portability**
>
> The only portable use of **signal**() is to set a signal's disposition to **SIG_DFL** or **SIG_IGN**. The semantics when using **signal**() to establish a signal handler vary across systems (and POSIX.1 explicitly permits this variation); **do not use it for this purpose.**
> POSIX.1 solved the portability mess by specifying **sigaction**(2), which provides explicit control of the semantics when a signal handler is invoked; use that interface instead of **signal**().

Check out the man page if you have time!

Exit on ctrl+c

```
void handler(int sig) {
    exit(0);
}

int main() {
    signal(SIGINT, handler);
    while (true) {
        sleep(1);
    }
    return 0;
}
```

Looks good! ✅

Count number of SIGCHLDs received

```c
static volatile int sigchld_count = 0;

void handler(int sig) {
    sigchld_count += 1;
}

int main() {
    signal(SIGCHLD, handler);
    const int num_processes = 10;
    for (int i = 0; i < num_processes; i++) {
        if (fork() == 0) {
            sleep(1);
            exit(0);
        }
    }
    while (waitpid(-1, NULL, 0) != -1) {}
    printf("All %d processes exited, got %d SIGCHLDs.\n",
        num_processes, sigchld_count);
    return 0;
}
```

Okay **if we were to use sigaction** ⚠️

Count number of running processes

```c
static volatile int running_processes = 0;

void handler(int sig) {
    while (waitpid(-1, NULL, WNOHANG) > 0) {
        running_processes -= 1;
    }
}

int main() {
    signal(SIGCHLD, handler);
    const int num_processes = 10;
    for (int i = 0; i < num_processes; i++) {
        if (fork() == 0) {
            sleep(1);
            exit(0);
        }
        running_processes += 1;
        printf("%d running processes\n", running_processes);
    }
    while(running_processes > 0) {
        pause();
    }
    printf("All processes exited! %d running processes\n", running_processes);
    return 0;
}
```

Not safe (concurrent use of running_processes) 🚫

Print on ctrl+c

```c
void handler(int sig) {
    printf("Hehe, not exiting!\n");
}

int main() {
    signal(SIGINT, handler);
    while (true) {
        printf("Looping...\n");
        sleep(1);
    }
    return 0;
}
```

Not safe!! 🚫

Print on ctrl+c

```c
void handler(int sig) {
    printf("Hehe, not exiting!\n");
}

int main() {
    signal(SIGINT, handler);
    while (true) {
        printf("Looping...\n");
        sleep(1);
    }
    return 0;
}
```

Not safe!! 🚫

```c
void print_hello(int sig) {
    printf("Hello world!\n");
}
```

```c
int main() {
    const char* message = "Hello world ";
    const size_t repeat = 1000;

    char *repeated_msg = malloc(repeat * strlen(message) + 2);
    for (int i = 0; i < repeat; i++) {
        strcpy(repeated_msg + (i * strlen(message)), message);
    }
    repeated_msg[repeat * strlen(message)] = '\n';
    repeated_msg[repeat * strlen(message) + 1] = '\0';

    signal(SIGUSR1, print_hello);
    if (fork() == 0) {
        pid_t parent_pid = getppid();
        while (true) {
            kill(parent_pid, SIGUSR1);
        }
        return 0;
    }

    while (true) {
        printf(repeated_msg);
    }

    free(repeated_msg);
    return 0;
}
```

./print ⌥⌘4

```
ld Hello world Hello world Hello world Hello world Hello world HelloHello world!
Hello world!
Hello world!
Hello world!
 world Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello wor
ld Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello world H
ello world Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello
 world Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello wor
ld Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello world H
ello world Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello
 world Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello wor
ld Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello world H
ello world Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello
 world Hello world Hello world Hello world Hello world Hello world HHello world!
Hello world!
Hello world!
Hello world!
ello world Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello
 world Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello wor
ld Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello world H
ello world Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello
 world Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello wor
ld Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello world H
ello world
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

reberhardt@ryan-mba: ~/Documents/Stanford/CS 110L/cs110l/lecture-examples/lecture8

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hell
o world Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello wo
rld Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello world
Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hell
o world Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello wo
rld Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello world
Hello world Hello world Hello world Hello world Hello world Hello world!
Hello world!
Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hell
o world Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello wo
rld Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello world
Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hello world Hell
o world Hello world Hello world Hello world Hello world Hello world Hello world
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
[1]    44332 abort      ./print
lecture8 git:(master) ✗
```

reberhardt@ryan-mba: ~/Documents/Stanford/CS 110L/cs110l/lecture-examples/lecture8

```
lecture8 git:(master) ✗ ./print
[1]    44156 illegal hardware instruction  ./print
```

Async-safe functions

- vfprintf is a 1787-line function!

```
/* Lock stream.  */
_IO_cleanup_region_start ((void (*) (void *)) &_IO_funlockfile, s);
_IO_flockfile (s);
```

- Apparently also does some other async-unsafe business
- You should avoid functions that use global state
  - Many functions do this, even if you may not realize it
  - malloc and free are not async-signal-safe!
- List of safe functions: http://man7.org/linux/man-pages/man7/signal-safety.7.html

What should we do?

Avoiding signal handling

- Anything substantial should not be done in a signal handler
- How can we handle signals, then?
- The "self-pipe" trick was invented in the early 90s:
  - Create a pipe
  - When you're awaiting a signal, read from the pipe (this will block until something is written to it)
  - In the signal handler, write a single byte to the pipe

Avoiding signal handling

- signalfd added official support for this hack

```c
int main(int argc, char *argv[]) {
    sigset_t mask;
    int sfd;
    struct signalfd_siginfo fdsi;
    ssize_t s;

    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGQUIT);

    /* Block signals so that they aren't handled
       according to their default dispositions */

    if (sigprocmask(SIG_BLOCK, &mask, NULL) == -1)
        handle_error("sigprocmask");

    sfd = signalfd(-1, &mask, 0);
    if (sfd == -1) handle_error("signalfd");
```

```c
for (;;) {
    s = read(sfd, &fdsi,
        sizeof(struct signalfd_siginfo));
    if (s != sizeof(struct signalfd_siginfo))
        handle_error("read");

    if (fdsi.ssi_signo == SIGINT) {
        printf("Got SIGINT\n");
    } else if (fdsi.ssi_signo == SIGQUIT) {
        printf("Got SIGQUIT\n");
        exit(EXIT_SUCCESS);
    } else {
        printf("Read unexpected signal\n");
    }
}
```

What about asynchronous signal handling?

- I thought part of the benefit of signal handlers was you can handle events asynchronously! (You can be doing work in your program, and quickly take a break to do something to handle a signal)
- Reading from a pipe or signalfd precludes concurrency: I’m either doing work, or reading to wait for a signal, but not both at the same time
- How can we address this?
  - Use threads
    - Can still have concurrency problems!
    - But we have more tools to reason about and control those problems
  - Use non-blocking I/O (week 8)

Ctrlc crate

- Rust has a ctrlc crate: register a function to be executed on ctrl+c (SIGINT)
- How does it work?
  - Creates a self-pipe
  - Installs a signal handler that writes to the pipe when SIGINT is received
  - Spawns a thread: loop { read from pipe; call handler function; }
- The Rust borrow checker prevents data races caused by concurrent access/modification from threads. If your handler function touches data in a racey way, the compiler will complain

Why is this different?

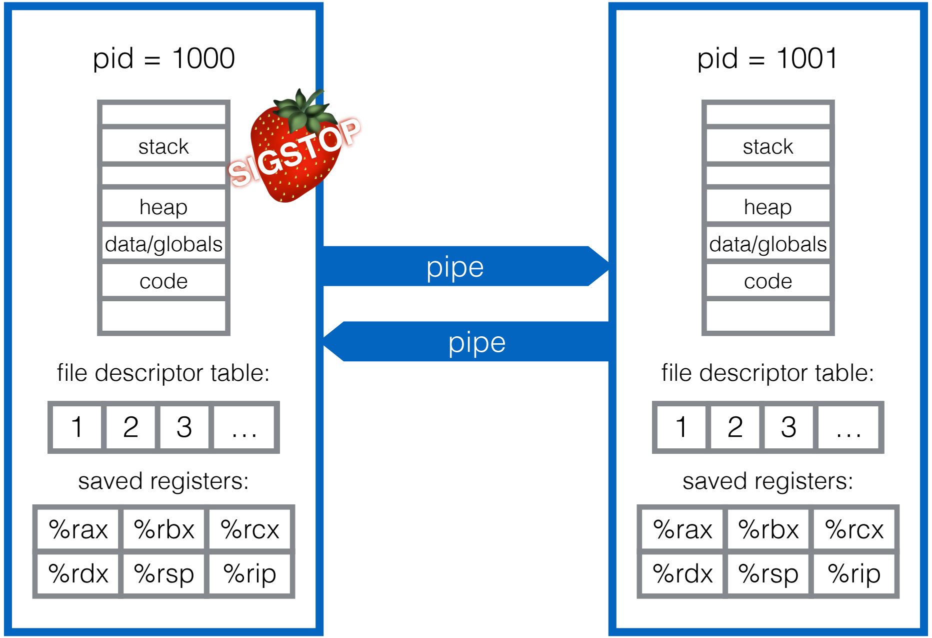
- printf from signal handler can deadlock:
  - printf from main body of code calls flock()
  - signal handler interrupts execution. printf from signal handler calls flock()
  - signal handler can’t continue until main code releases lock, but main code can’t continue until the signal handler exits
- printf from threads are safe:
  - printf from main thread calls flock()
  - printf from signal handling thread calls flock() and is blocked
  - printf from main thread finishes
  - printf from signal handling thread finishes
- malloc() calls (including the ones printf makes) work similarly.

Why is this different?

- Threads and signal handlers have the same concurrency problems
- But the scheduling of code is completely different
- Threads:
  - Multiple (usually) equal-priority threads of execution that constantly swap on the processor
  - Can use locks to protect data
- Signal handlers:
  - Handler will completely preempt all other code and hog the CPU until it finishes
  - Can’t use locks or any other synchronization primitives
    - In fact, signal handlers should avoid all kinds of blocking! (Why?)
  - Consequently, signal handlers play very poorly with library code. Libraries don’t know what signal handlers you have installed or what those signal handlers do, so they can’t disable signal handling to protect themselves from concurrency problems

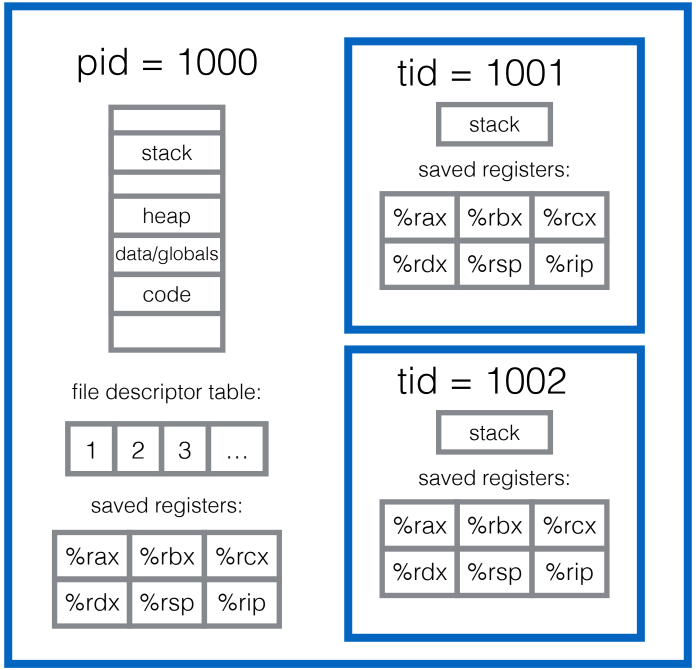
Google Chrome

Processes



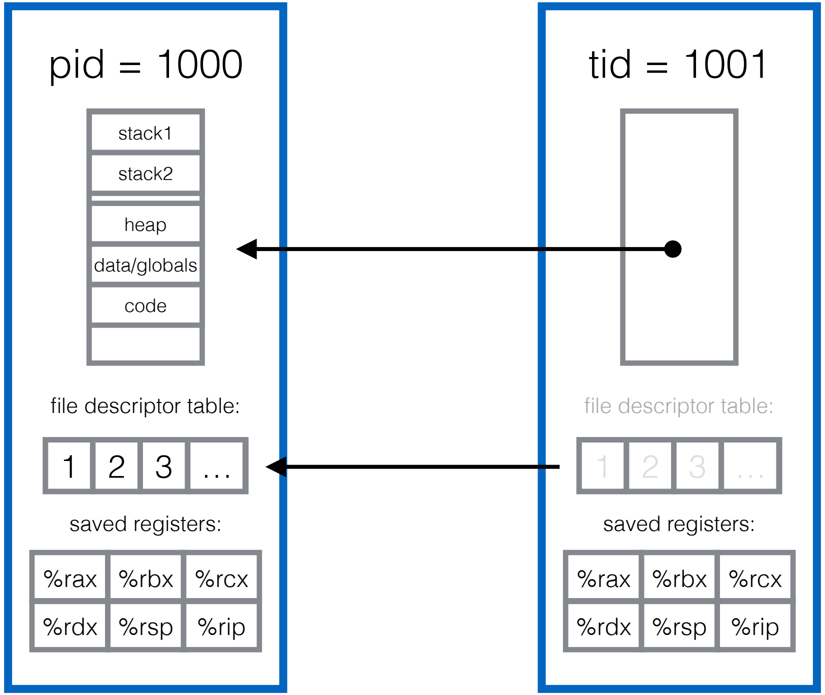
Processes can synchronize using signals and pipes

Threads



Threads are similar to processes; they have a separate stack and saved registers (and a handful of other separated things). But they share most resources across the process

Threads



Under the hood, a thread gets its own “process control block” and is scheduled independently, but it is linked to the process that spawned it

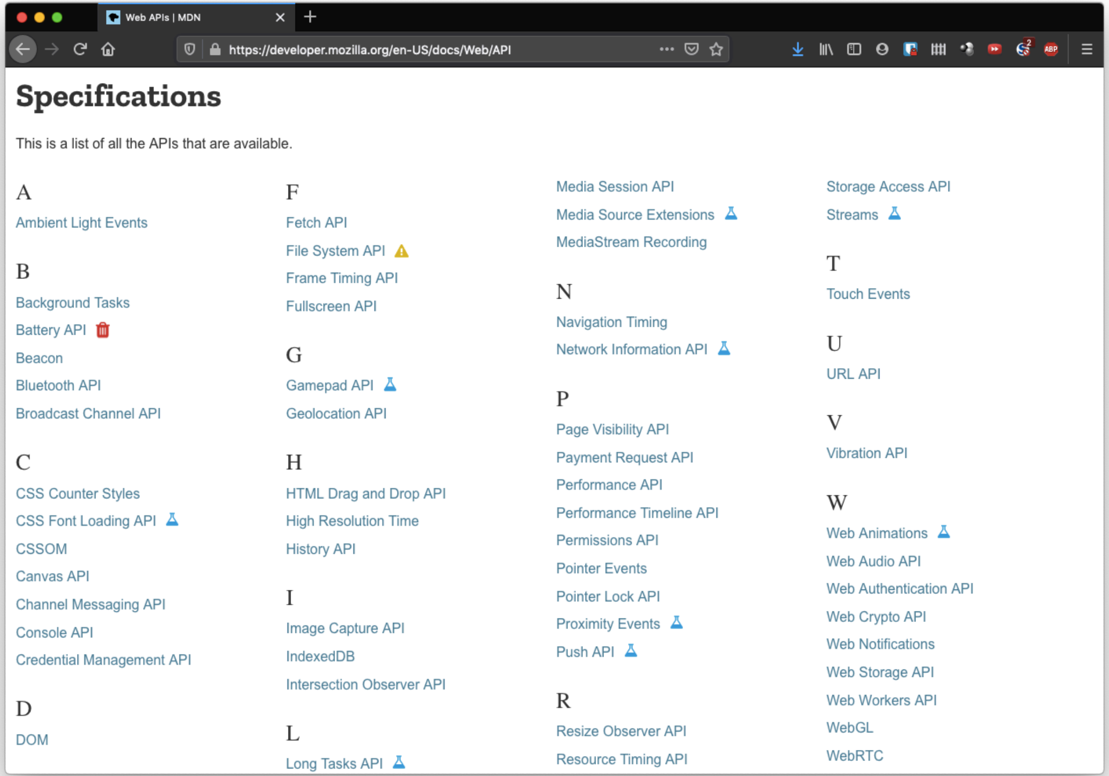
Considerations when designing a browser

- Speed
- Memory usage
- Battery/CPU usage
- Ease of development
- Security, stability

Considerations when designing a browser

- Speed
  - Typically faster to share memory and to use lightweight synchronization primitives
- Memory usage
  - Processes use more memory
- Battery/CPU usage
  - Threads incur less context switching overhead
- Ease of development
  - Communication is WAY easier using threads
  - (That being said, bugs caused by multithreading are extremely hard to track down)
- Security, stability
  - Multiprocessing provides isolation. Multithreading does not.

Modern browsers are essentially operating systems


Web APIs | MDN

https://developer.mozilla.org/en-US/docs/Web/API

Specifications

This is a list of all the APIs that are available.

A
Ambient Light Events

B
Background Tasks
Battery API
Beacon
Bluetooth API
Broadcast Channel API

C
CSS Counter Styles
CSS Font Loading API
CSSOM
Canvas API
Channel Messaging API
Console API
Credential Management API

D
DOM

F
Fetch API
File System API
Frame Timing API
Fullscreen API

G
Gamepad API
Geolocation API

H
HTML Drag and Drop API
High Resolution Time
History API

I
Image Capture API
IndexedDB
Intersection Observer API

L
Long Tasks API

Media Session API
Media Source Extensions
MediaStream Recording

N
Navigation Timing
Network Information API

P
Page Visibility API
Payment Request API
Performance API
Performance Timeline API
Permissions API
Pointer Events
Pointer Lock API
Proximity Events
Push API

R
Resize Observer API
Resource Timing API

Storage Access API
Streams

T
Touch Events

U
URL API

V
Vibration API

W
Web Animations
Web Audio API
Web Authentication API
Web Crypto API
Web Notifications
Web Storage API
Web Workers API
WebGL
WebRTC

https://developer.mozilla.org/en-US/docs/Web/API

Modern browsers are essentially operating systems

- Storage APIs
- Concurrency APIs
- Hardware APIs (e.g. communicate with MIDI devices, even GPU)
- Run assembly
- Run Windows 95: https://win95.ajf.me/

Motivation for Chrome

> *It's nearly impossible to build a rendering engine that never crashes or hangs. It's also nearly impossible to build a rendering engine that is perfectly secure.*
>
> *In some ways, the state of web browsers around 2006 was like that of the single-user, co-operatively multi-tasked operating systems of the past. As a misbehaving application in such an operating system could take down the entire system, so could a misbehaving web page in a web browser. All it took is one browser or plug-in bug to bring down the entire browser and all of the currently running tabs.*
>
> *Modern operating systems are more robust because they put applications into separate processes that are walled off from one another. A crash in one application generally does not impair other applications or the integrity of the operating system, and each user's access to other users' data is restricted.*

https://www.chromium.org/developers/design-documents/multi-process-architecture

Motivation for Chrome

> *Compromised renderer processes (also known as "arbitrary code execution" attacks in the renderer process) need to be explicitly included in a browser's security threat model. We assume that determined attackers will be able to find a way to compromise a renderer process, for several reasons:*
>
> - *Past experience suggests that potentially exploitable bugs will be present in future Chrome releases. There were 10 potentially exploitable bugs in renderer components in M69, 5 in M70, 13 in M71, 13 in M72, 15 in M73. This volume of bugs holds steady despite years of investment into developer education, fuzzing, Vulnerability Reward Programs, etc. Note that this only includes bugs that are reported to us or are found by our team.*
> - *Security bugs can often be made exploitable: even 1-byte buffer overruns can be turned into an exploit.*
> - *Deployed mitigations (like ASLR or DEP) are not always effective.*

https://www.chromium.org/Home/chromium-security/site-isolation

Motivation for Chrome

Compromised renderer processes (also known as "arbitrary code execution" attacks in the renderer process) need to be explicitly included in a browser's security threat model. We assume that ***determined attackers will be able to find a way to compromise a renderer process****, for several reasons:*

- *Past experience suggests that potentially exploitable bugs will be present in future Chrome releases. There were 10 potentially exploitable bugs in renderer components in M69, 5 in M70, 13 in M71, 13 in M72, 15 in M73. This volume of bugs holds steady despite years of investment into developer education, fuzzing, Vulnerability Reward Programs, etc. Note that this only includes bugs that are reported to us or are found by our team.*
- *Security bugs can often be made exploitable: even 1-byte buffer overruns can be turned into an exploit.*
- *Deployed mitigations (like ASLR or DEP) are not always effective.*

https://www.chromium.org/Home/chromium-security/site-isolation

Motivation for Chrome

> *Compromised renderer processes (also known as "arbitrary code execution" attacks in the renderer process) need to be explicitly included in a browser's security threat model. We assume that determined attackers will be able to find a way to compromise a renderer process, for several reasons:*
>
> - ***Past experience suggests that potentially exploitable bugs will be present in future Chrome releases.*** *There were 10 potentially exploitable bugs in renderer components in M69, 5 in M70, 13 in M71, 13 in M72, 15 in M73. This volume of bugs holds steady despite years of investment into developer education, fuzzing, Vulnerability Reward Programs, etc. Note that this only includes bugs that are reported to us or are found by our team.*
> - *Security bugs can often be made exploitable: even 1-byte buffer overruns can be turned into an exploit.*
> - *Deployed mitigations (like ASLR or DEP) are not always effective.*

https://www.chromium.org/Home/chromium-security/site-isolation

Motivation for Chrome

> *Compromised renderer processes (also known as "arbitrary code execution" attacks in the renderer process) need to be explicitly included in a browser's security threat model. We assume that determined attackers will be able to find a way to compromise a renderer process, for several reasons:*
>
> - *Past experience suggests that potentially exploitable bugs will be present in future Chrome releases. There were 10 potentially exploitable bugs in renderer components in M69, 5 in M70, 13 in M71, 13 in M72, 15 in M73.* ***This volume of bugs holds steady despite years of investment into developer education, fuzzing, Vulnerability Reward Programs, etc.*** *Note that this only includes bugs that are reported to us or are found by our team.*
> - *Security bugs can often be made exploitable: even 1-byte buffer overruns can be turned into an exploit.*
> - *Deployed mitigations (like ASLR or DEP) are not always effective.*

https://www.chromium.org/Home/chromium-security/site-isolation

Motivation for Chrome

> *Compromised renderer processes (also known as "arbitrary code execution" attacks in the renderer process) need to be explicitly included in a browser's security threat model. We assume that determined attackers will be able to find a way to compromise a renderer process, for several reasons:*
>
> - *Past experience suggests that potentially exploitable bugs will be present in future Chrome releases. There were 10 potentially exploitable bugs in renderer components in M69, 5 in M70, 13 in M71, 13 in M72, 15 in M73. This volume of bugs holds steady despite years of investment into developer education, fuzzing, Vulnerability Reward Programs, etc.* ***Note that this only includes bugs that are reported to us or are found by our team.***
> - *Security bugs can often be made exploitable: even 1-byte buffer overruns can be turned into an exploit.*
> - *Deployed mitigations (like ASLR or DEP) are not always effective.*

https://www.chromium.org/Home/chromium-security/site-isolation

Motivation for Chrome

Compromised renderer processes (also known as "arbitrary code execution" attacks in the renderer process) need to be explicitly included in a browser's security threat model. We assume that determined attackers will be able to find a way to compromise a renderer process, for several reasons:

- *Past experience suggests that potentially exploitable bugs will be present in future Chrome releases. There were 10 potentially exploitable bugs in renderer components in M69, 5 in M70, 13 in M71, 13 in M72, 15 in M73. This volume of bugs holds steady despite years of investment into developer education, fuzzing, Vulnerability Reward Programs, etc. Note that this only includes bugs that are reported to us or are found by our team.*
- ***Security bugs can often be made exploitable**: even 1-byte buffer overruns can be turned into an exploit.*
- *Deployed mitigations (like ASLR or DEP) are not always effective.*

https://www.chromium.org/Home/chromium-security/site-isolation

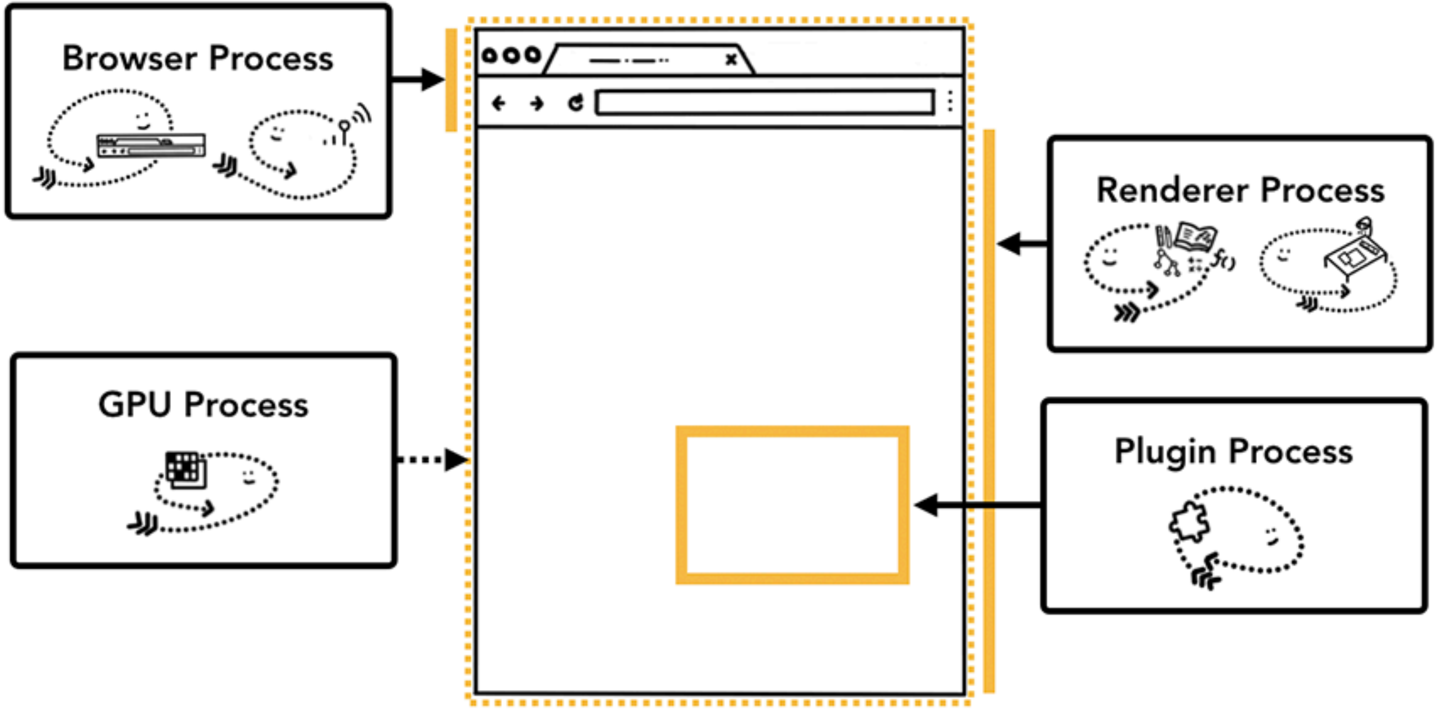
Motivation for Chrome

> *Compromised renderer processes (also known as "arbitrary code execution" attacks in the renderer process) need to be explicitly included in a browser's security threat model. We assume that determined attackers will be able to find a way to compromise a renderer process, for several reasons:*
>
> - *Past experience suggests that potentially exploitable bugs will be present in future Chrome releases. There were 10 potentially exploitable bugs in renderer components in M69, 5 in M70, 13 in M71, 13 in M72, 15 in M73. This volume of bugs holds steady despite years of investment into developer education, fuzzing, Vulnerability Reward Programs, etc. Note that this only includes bugs that are reported to us or are found by our team.*
> - *Security bugs can often be made exploitable: even 1-byte buffer overruns can be turned into an exploit.*
> - ***Deployed mitigations (like ASLR or DEP) are not always effective.***

https://www.chromium.org/Home/chromium-security/site-isolation

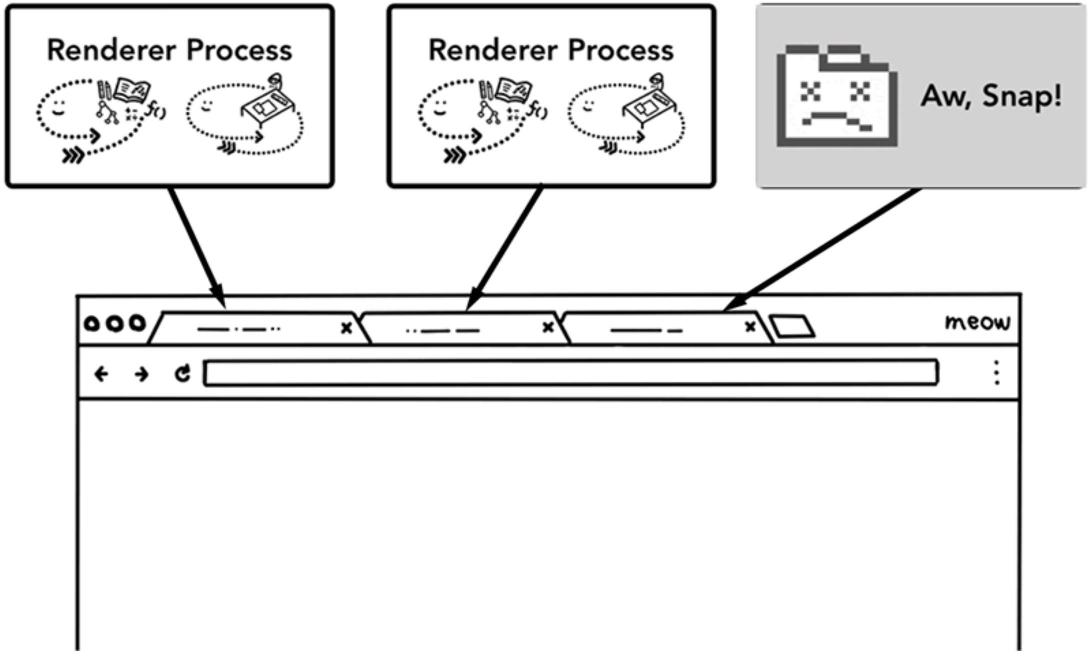
Aside: What does Firefox’s architecture look like?

Chrome architecture



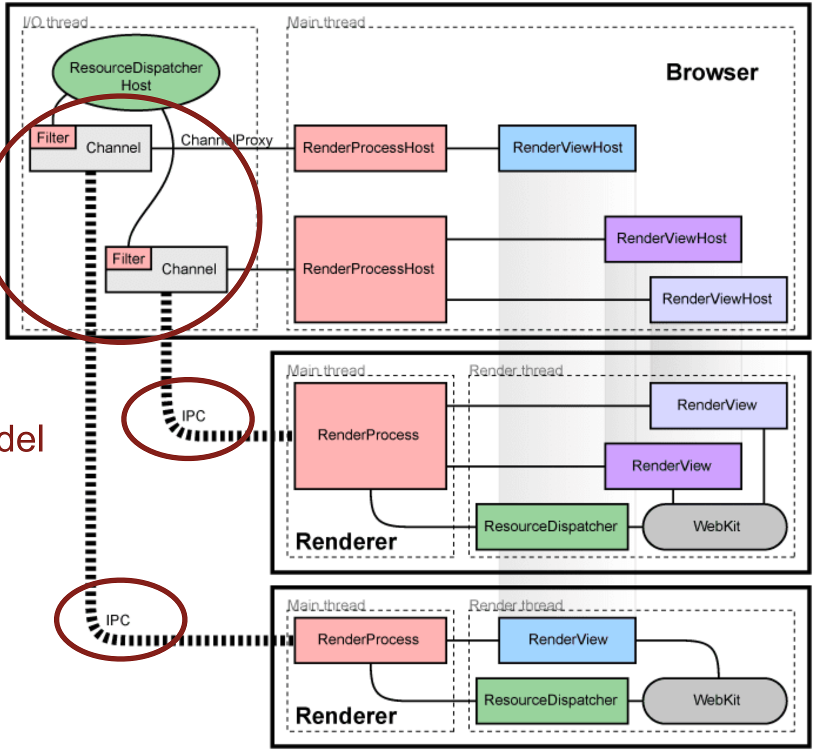
REALLY CUTE diagrams from https://developers.google.com/web/updates/2018/09/inside-browser-part1
(great read!)

Chrome architecture



REALLY CUTE diagrams from https://developers.google.com/web/updates/2018/09/inside-browser-part1
(great read!)

Chrome architecture



IPC channels = pipes

Message passing model

Events (e.g. click, keystroke, etc) are relayed through these pipes! No signals

Sandboxed processes: no access to network, filesystem, etc

If there is embedded content, may use multiple threads to render that content and manage communication between frames

https://www.chromium.org/developers/design-documents/multi-process-architecture (slightly out of date)

Not good enough

- What does all this work buy us?
  - Isolation between tabs
  - Isolation between (potentially malicious) websites and the host
- What does it *not* buy us?
  - Isolation between resources *within* a tab

Not good enough

Same-origin policy: www.evil.com can embed bank.com, but cannot interact with bank.com or see its data

Not good enough

- Site Isolation Project (2015-2019) aimed to put resources for different origins in different processes
- Extremely difficult undertaking. Cross-frame communication is common (JS postMessage API), and embedded frames need to share render buffers
  - Involved rearchitecting the most core parts of Chrome
- Became especially important in Jan 2018: Spectre and Meltdown
  - When the hardware fails to uphold its guarantees, JS can read arbitrary process memory (even kernel memory, and even if your software has no bugs)!
- Paper/video: https://www.usenix.org/conference/usenixsecurity19/presentation/reis

Anatomy of a sandbox escape

- https://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html (2012 but it’s more accessible than some other writeups)
  - First exploit chains together *six bugs* to escape the sandbox
  - Second one uses *ten(!!)*
- https://googleprojectzero.blogspot.com/2019/04/virtually-unlimited-memory-escaping.html (2019)

More relevant reading

- How Chrome does fork():
  http://neugierig.org/software/chromium/notes/2011/08/zygote.html
  Fun related bug report: https://bugs.chromium.org/p/chromium/issues/detail?id=35793
  What steps will reproduce the problem?
  1. Develop a webapp, use chrome's devtools, minding your own business
  2. In the meantime, let chrome silently autoupdate in the background

  What is the expected result?
  Devtools continue working

  What happens instead?
  Devtools break after refreshing the page after the autoupdate happened.