Distributed Trace Compare
Technical and practical point of view

ICPE
2024

Maryam Ekhlasi
Nasser Ezzati-Jivan

May 2024

Who we are?

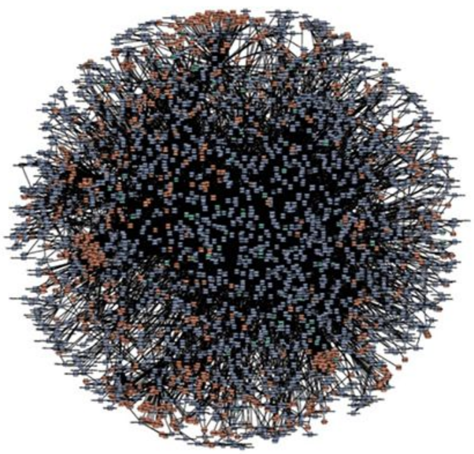
Nasser Ezzati-Jivan

- From Brock University
- Assistant Professor
- Professor, software engineer, and team leader since 2008

Maryam Ekhlasi

- From Polytechnique Montreal University, DORSAL Lab.
- Ph.D. Candidate
- Software Performance researcher
- Software engineer since 2011

Goal



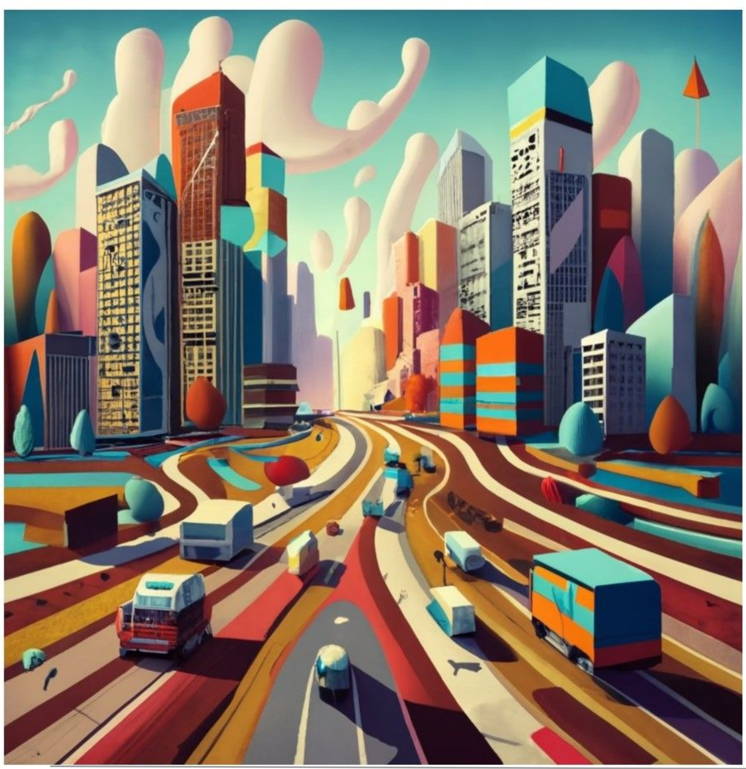
Agenda

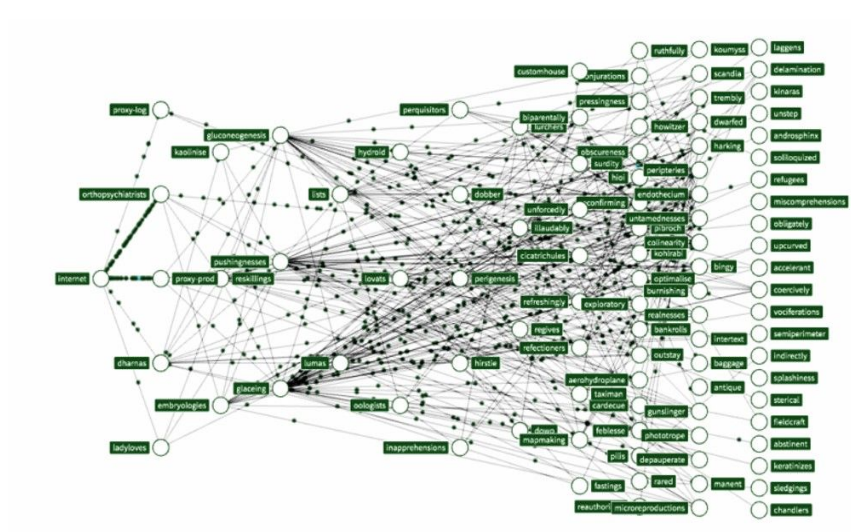
1. Introduction (5 min)
2. Motivations (5 min)
3. Basic concepts (60 min)
   - Low-level Tracing, End-to-end Tracing, Debugging, Profiling, Logging, Monitoring
   - Kernel/user space
   - Process/ thread states
1. Detecting/Diagnosing and fixing real performance problems (60 min)
   - Introducing real performance problems
   - Introducing Differential flame graph
   - Detecting performance problems
   - Diagnosing performance problems
2. Hands-on exercises (30 min)
   - Installing TraceCompass
   - Collecting Traces with LTTng
3. Conclusion (5 min)
4. Questions and Answers (5 min)

Special Thanks!

- Professor Michel Dagenais
  - Full Professor at Polytechnique Montreal University
- Fatemeh Faraji Daneshgar
  - Research Associate at Polytechnique Montreal University
  - Developer of Differential Flame Graph in TraceCompas
- TraceCompass Team
- EfficiOS Team

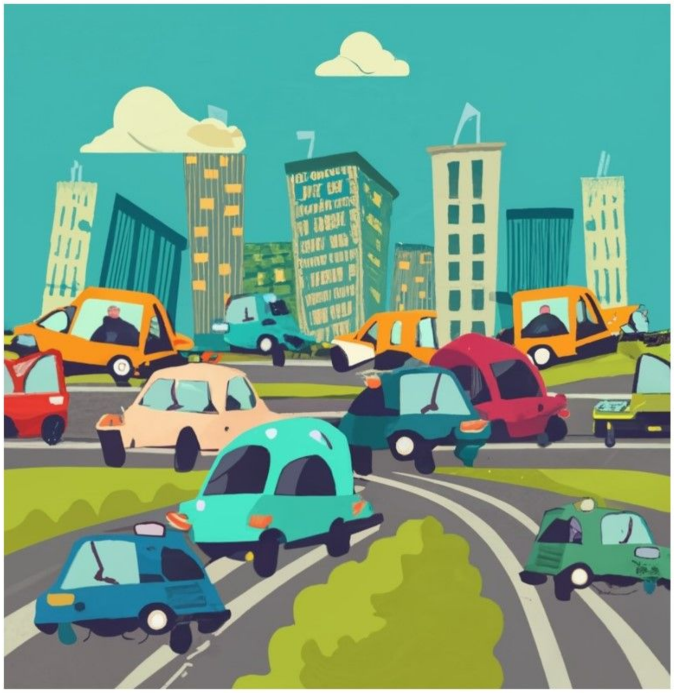
Welcome to Microservice City!

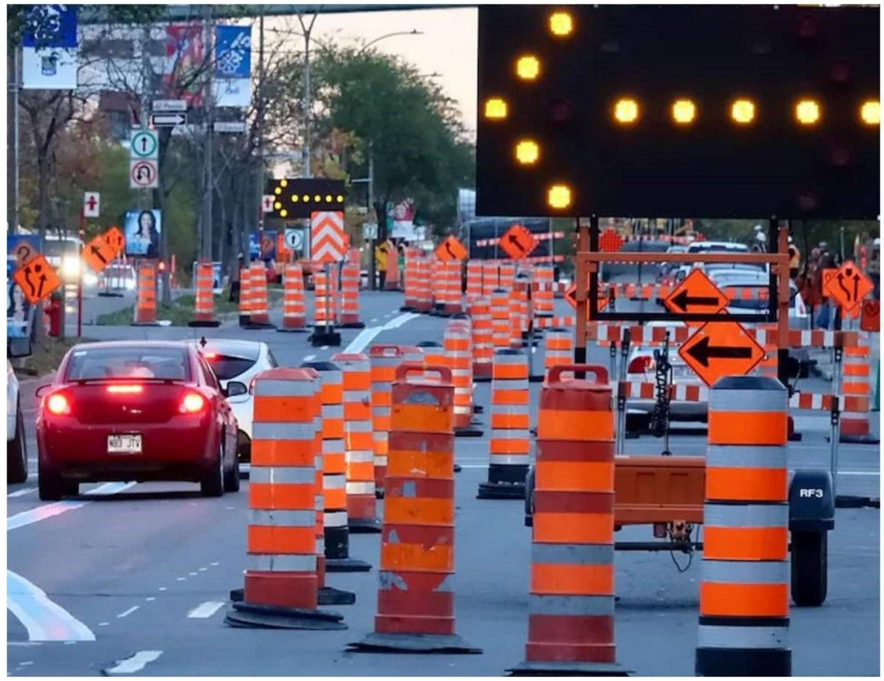




Reference: https://www.honeycomb.io/microservices

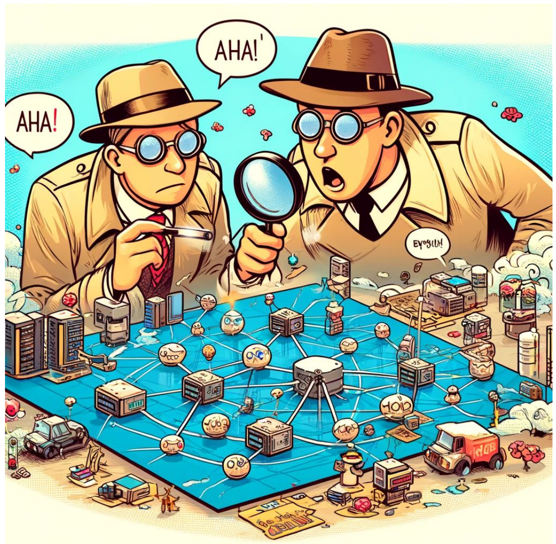
latency issue!



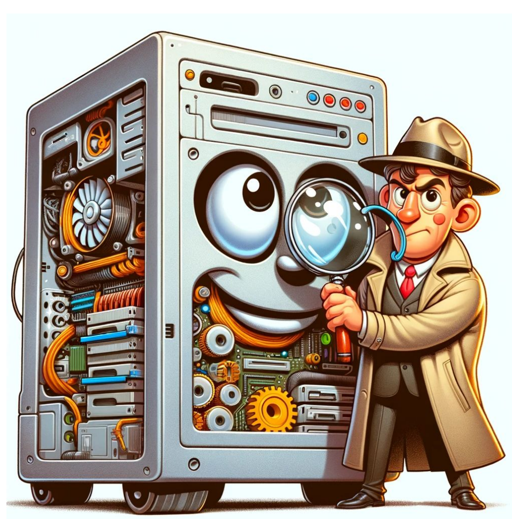


Basic Concepts!

Question: Have you ever wondered how developers track the behavior of distributed applications in real-time, especially in complex systems?



Question: What technique allows software developers to analyze and monitor the detailed operations of a system, including interactions with the operating system, to enhance performance and diagnose issues?!



Levels of observability

- Debugging
- Logging
- Profiling
- Tracing
  - Low-level Tracing
  - High-level Tracing
- Monitoring

Debugging

- The process of finding errors in source code!
- Performed in the development environment.

```
def calculate_average(numbers):
    total_sum = sum(numbers)
    count = len(numbers)
    average = total_sum / count  # Potential division by zero error
    return average

# List of numbers
numbers = []

# Calculate the average
print("The average is:", calculate_average(numbers))
```

Throw a division by zero error!!

Logging

- Recording information during the execution time.
- Levels of logging
  - DEBUG
  - INFO
  - WARNING
  - ERROR
  - FATAL

```python
import logging

# Configure logging
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s -

def process_transaction(transaction_id, amount):
    logging.info(f"Starting transaction for ID: {transaction_id} with amount:
    try:
        # Simulate a transaction processing
        if amount <= 0:
            raise ValueError("Amount must be positive")
        # Simulate successful transaction processing
        logging.info(f"Transaction {transaction_id} processed successfully")
    except ValueError as e:
        logging.error(f"Error processing transaction {transaction_id}: {e}")

# Example of processing transactions
process_transaction('TX1001', 500)
process_transaction('TX1002', -150)
```

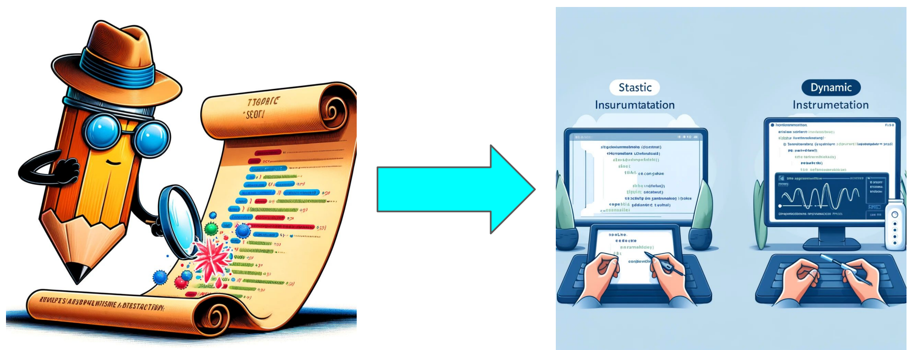

Profiling

- Statistical summary of observed events.
- Where is performance lost?
- Won't report the reason!

```python
import cProfile
import random
import time

def process_data_fast():
    time.sleep(1)  # Simulates processing time
    return sum(random.sample(range(10000), 1000))

def process_data_medium():
    time.sleep(2)  # Simulates more processing time
    return sum(random.sample(range(10000), 5000))

def process_data_slow():
    time.sleep(3)  # Simulates even more processing time
    data = [random.randint(1, 100) for _ in range(10000)]
    return sum(data)

def main():
    fast_result = process_data_fast()
    medium_result = process_data_medium()
    slow_result = process_data_slow()
    print(f"Fast: {fast_result}, Medium: {medium_result}, Slow: {slow_result}")

if __name__ == "__main__":
    profiler = cProfile.Profile()
    profiler.enable()
    main()
    profiler.disable()
    profiler.print_stats(sort='time')
```

Low-level Tracing!

- Printf!
- Every location in the code that we want to trace is referred to as a tracepoint. The process of inserting tracepoints into the code is known as instrumentation and each time a tracepoint is executed, it is generated an event.
- Instrumentation
  - Static instrumentation
  - Dynamic instrumentation



Low-level Tracing!

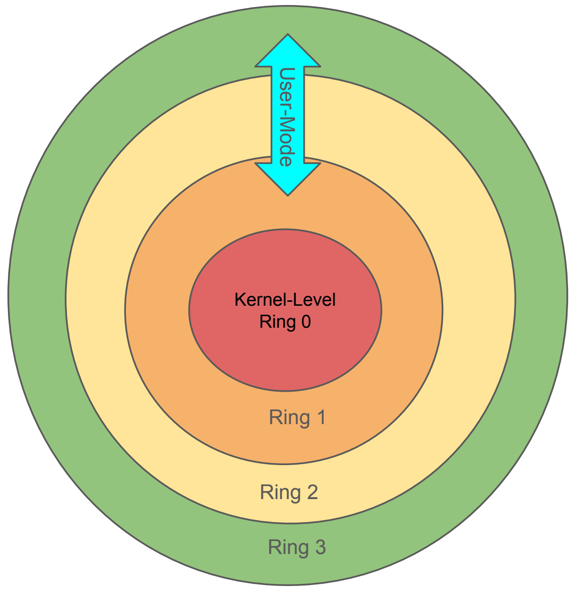
```c
#include <stdio.h>

#define MAX 5

int calculate_sum(int* array) {
    int total = 0;

    printf("Starting calculation");                // <-- tracepoint
    for (int i = 0; i < MAX; i++) {
        total += array[i];
        printf("Adding index %d, total now %d", i, total);  // <-- tracepoint
    }
    printf("Final total: %d", total);              // <-- tracepoint

    return total;
}

int main() {
    int numbers[MAX] = {1, 2, 3, 4, 5};
    calculate_sum(numbers);
    return 0;
}
```

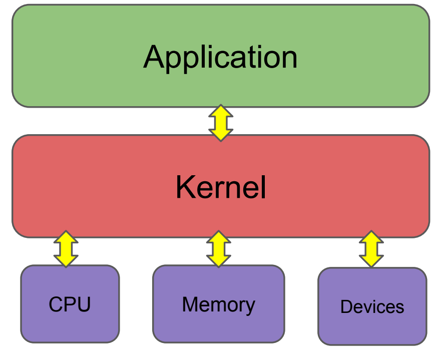
```
$ ./myprog
Starting calculation                  // <-- event
Adding index 0, total now 1           // <-- event
Adding index 1, total now 3           // <-- event
Adding index 2, total now 6           // <-- event
Adding index 3, total now 10          // <-- event
Adding index 4, total now 15          // <-- event
Final total: 15                       // <-- event
```

Two Levels of low-level tracing

User-space Tracing

Kernel-Level Tracing





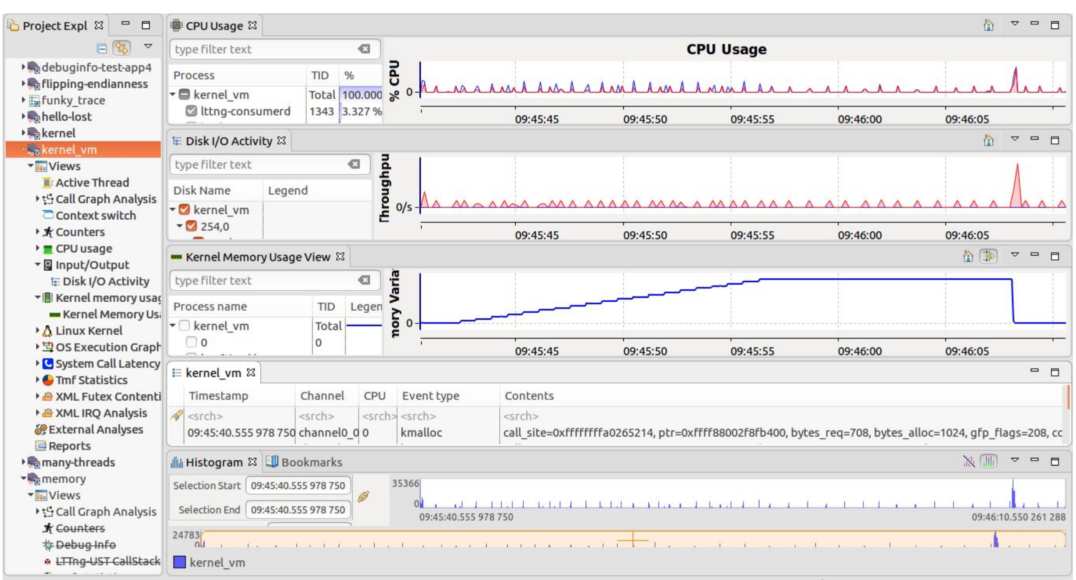
Tracing VS Logging

- Tracers are designed to record much lower-level events that occur much more frequently than log messages.
- Logging is more appropriate for a very high-level analysis of less frequent events: user accesses, exceptional conditions (errors and warnings, for example), database transactions, instant messaging communications, and such. Simply put, logging is one of the many use cases that can be satisfied with tracing.

Competing software tracers (low-level tracers)

- LTTng
- Dtrace4linux
- eBPF
- Ftrace
- Perf
- Strace
- Sysdig
- SystemTap

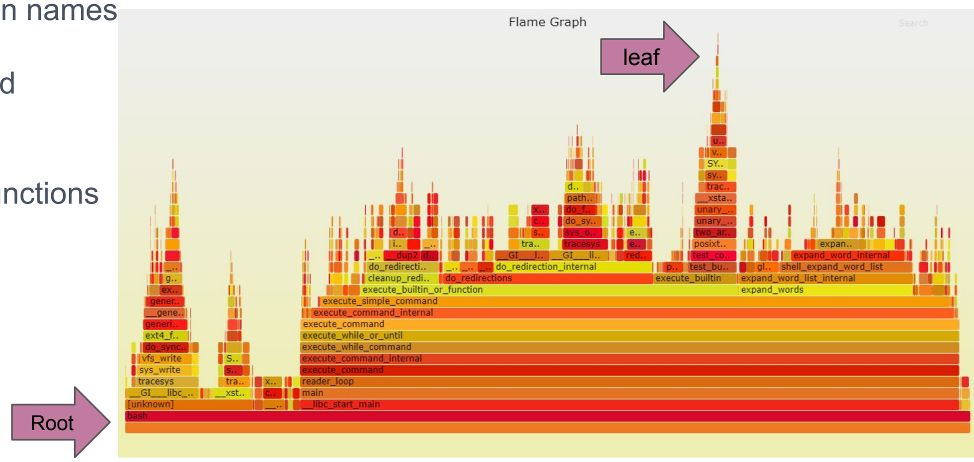
Low-level Trace Visualization Tool



Visualization Approaches

Flame Graph

- The horizontal axis of the call stack collection represents the function names
- Alphabetical arrangement of function names from left to right
- Combining identical functions placed side by side horizontally
- The width of each box shows the appearance frequency of that functions
- Random colors



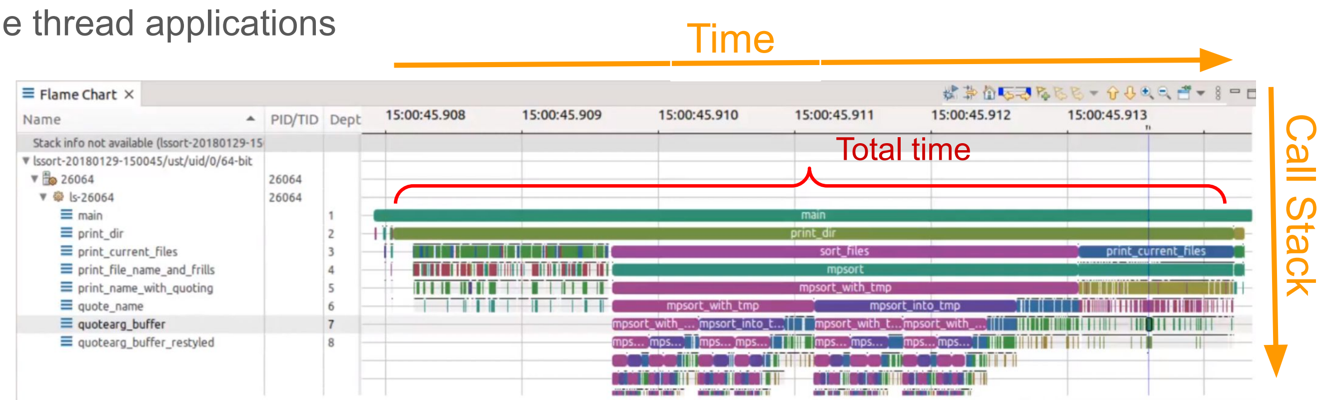
Reference:https://github.com/brendangregg/FlameGraph

Visualization Approaches

Flame Chart

The self time of a function is indicated by the difference in width between its frame and the frames directly beneath it

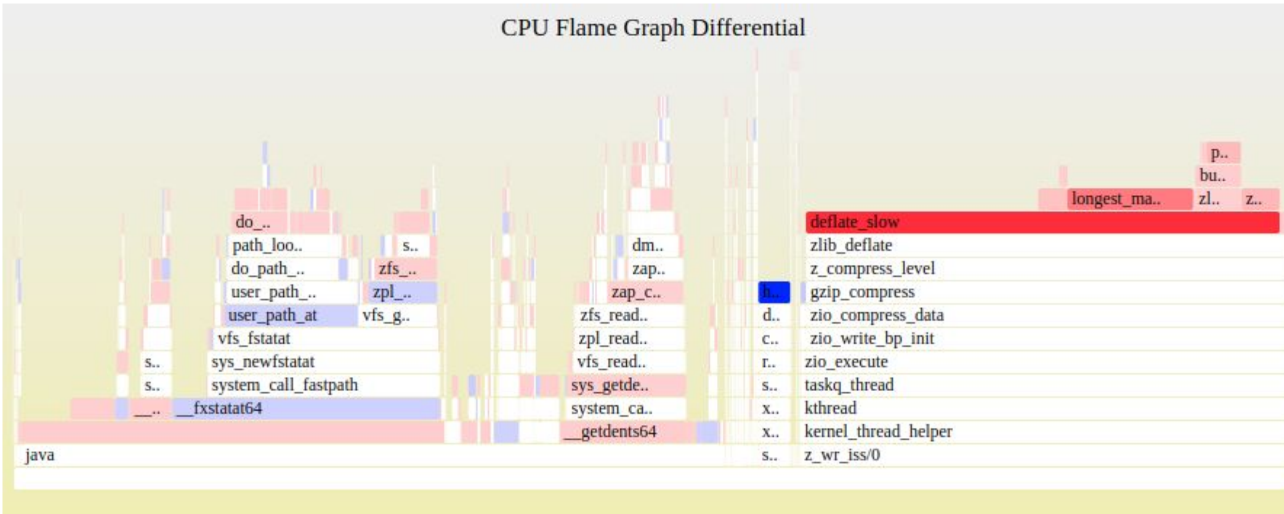
- Time on the horizontal axis
- Call stack on the vertical axis
- Single thread applications



Reference: https://archive.eclipse.org/tracecompass/doc/stable/org.eclipse.tracecompass.doc.user/LTTng-UST-Analyses.html

Visualization Approaches

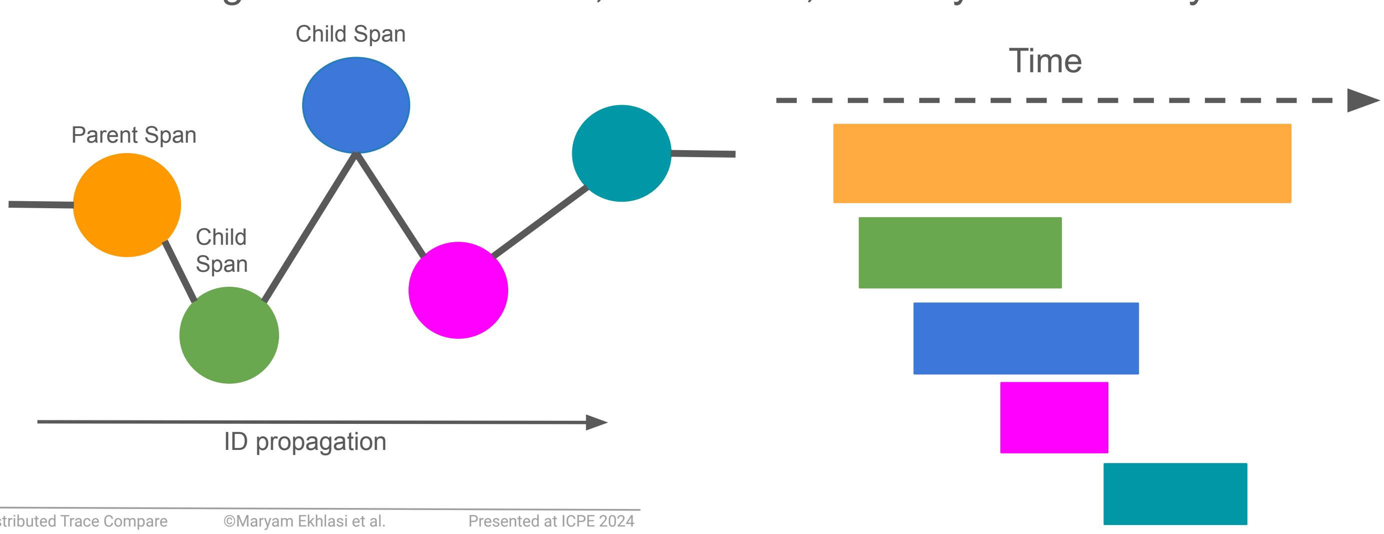
Differential Flame Graph



Reference: https://www.brendangregg.com/blog/2014-11-09/differential-flame-graphs.html

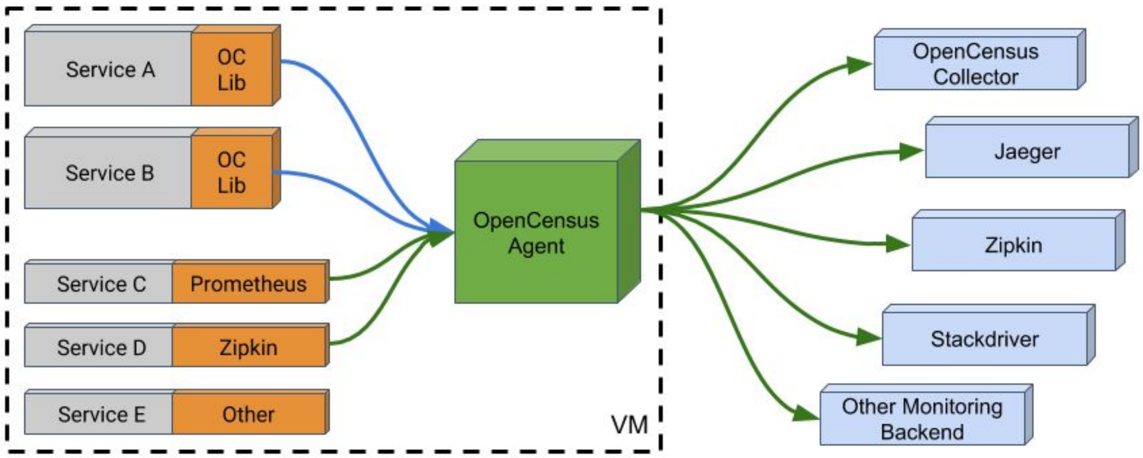
Distributed Tracing (High-level Tracing)

- End-to-end tracing!
- Technique for monitoring application requests as they move from frontend devices through backend services, databases, and any intermediary services.



Distributed End-to-end Tracing Standards

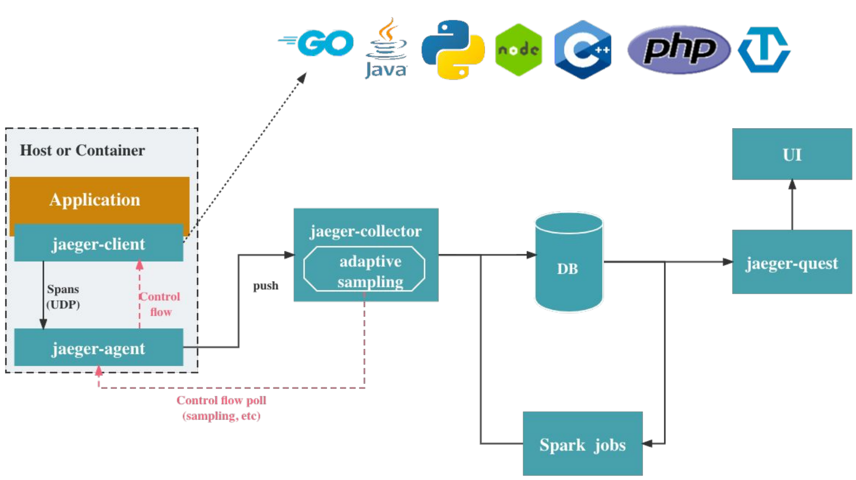
1. OpenCensus



Reference: https://opencensus.io/service/components/agent/

Distributed End-to-end Tracing Standards

2. OpenTracing



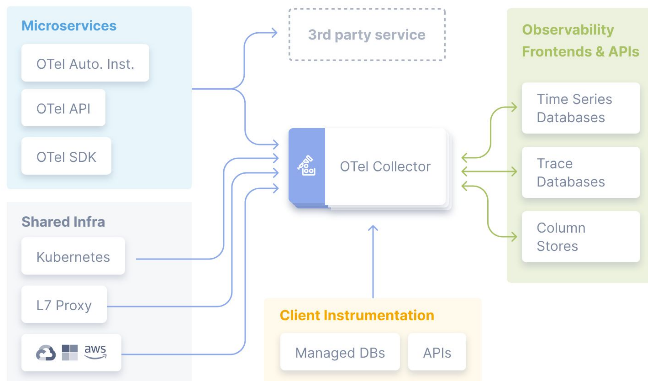
Reference: https://www.jaegertracing.io/docs/1.33/architecture/

Distributed End-to-end Tracing Standards

3. OpenTelemetry

OPENTRACING + OpenCensus = OpenTelemetry

CLOUD NATIVE COMPUTING FOUNDATION



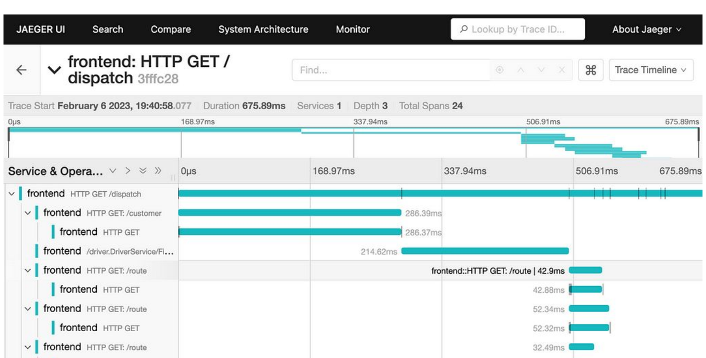
Reference: https://opentelemetry.io/docs/

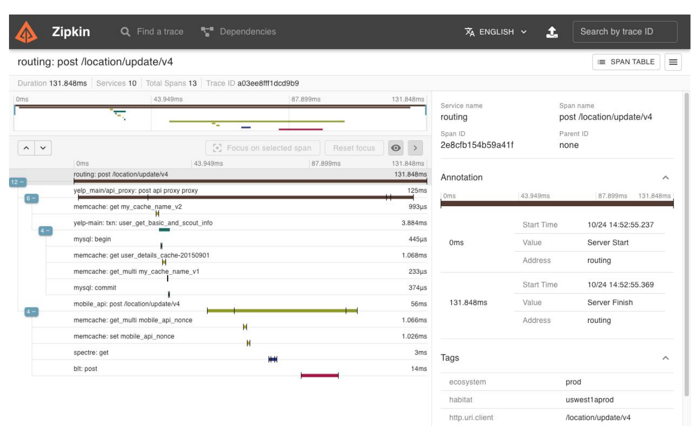
High-level Trace Visualization Tools

JAEGER

ZIPKIN

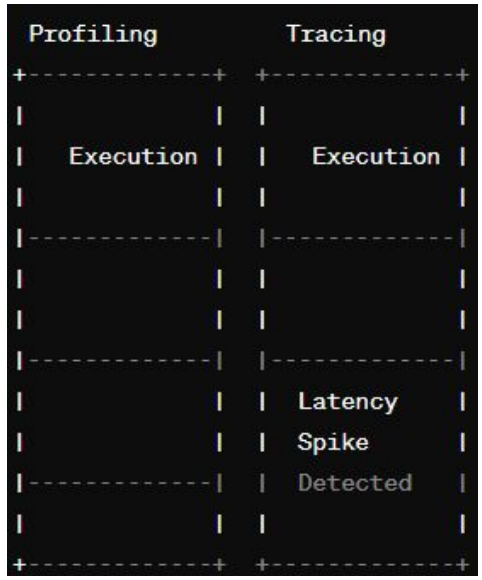
- Visualizing distributed requests
- Distributed context propagation
- Span's parent-child relationships





Power of Tracing!

- Over Profiling
- Over Debugging
- Over Logging
- Over Monitoring



Software Performance!

Deadlines!

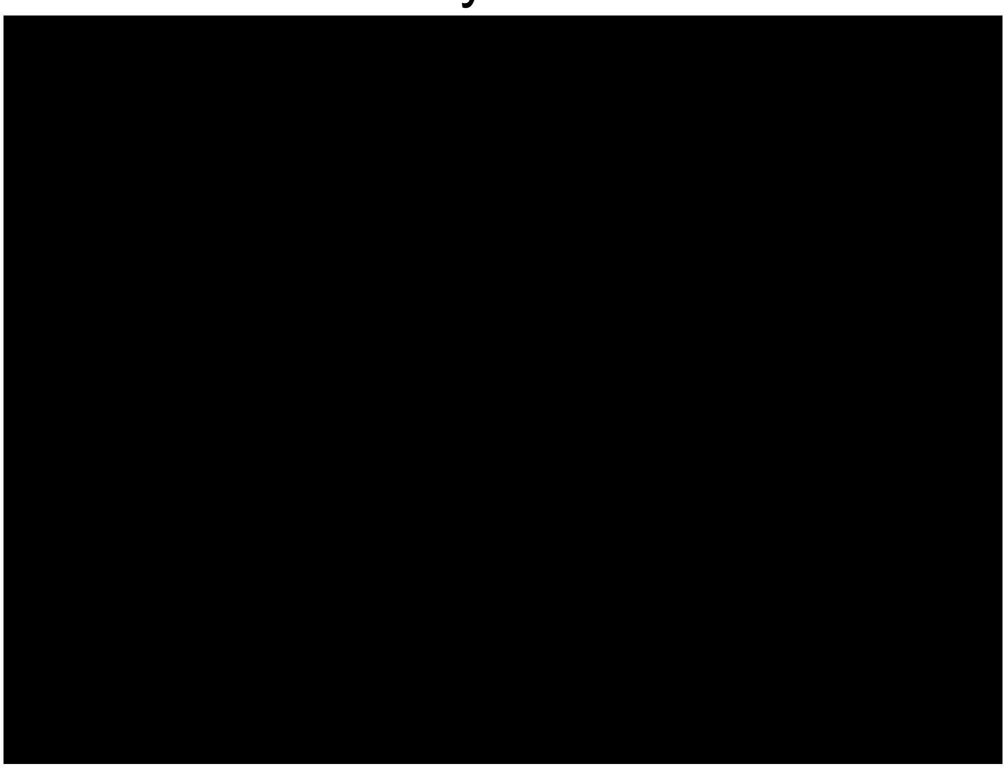
Fast!

Limited Machine Resources!

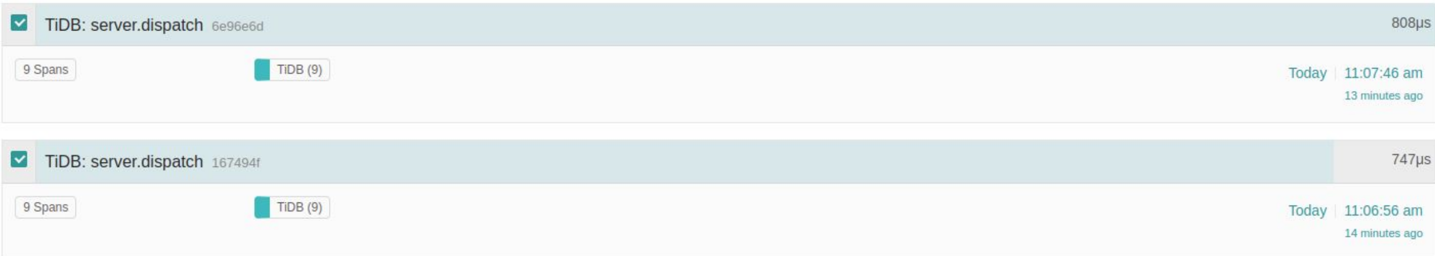
What is Software Performance

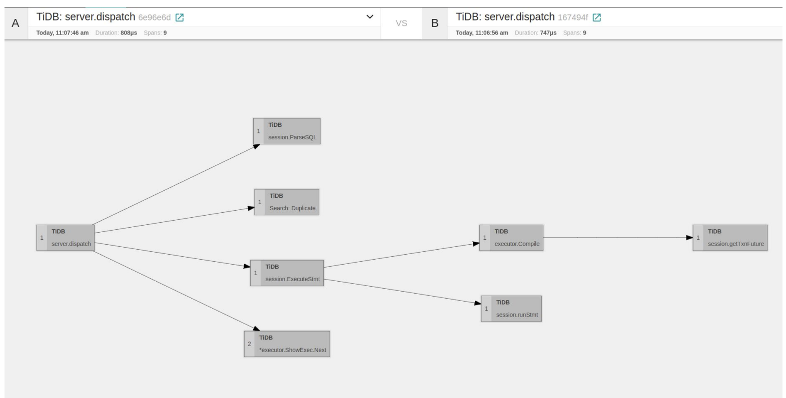
- Software Aspects
  - **Code efficiency**
  - Caching strategy
  - Architecture and Design
  - Concurrency and Parallelism
- Hardware Aspects
  - Processor Speed Core Count
  - Memory (RAM)
  - Storage (HDD/SSD)
  - Network Speed
- Other Factors

Motivation! Code Efficiency!

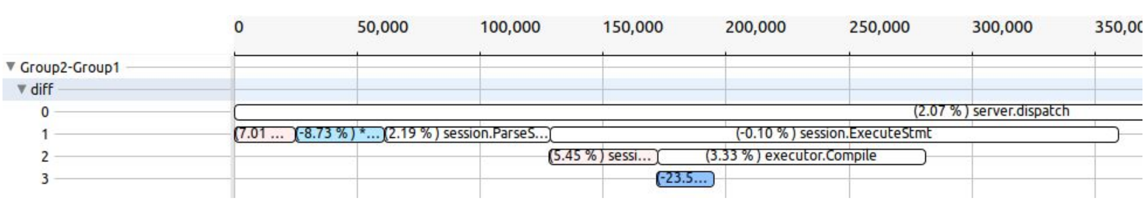


Executions Differences!





Detective DTraComp Enter!



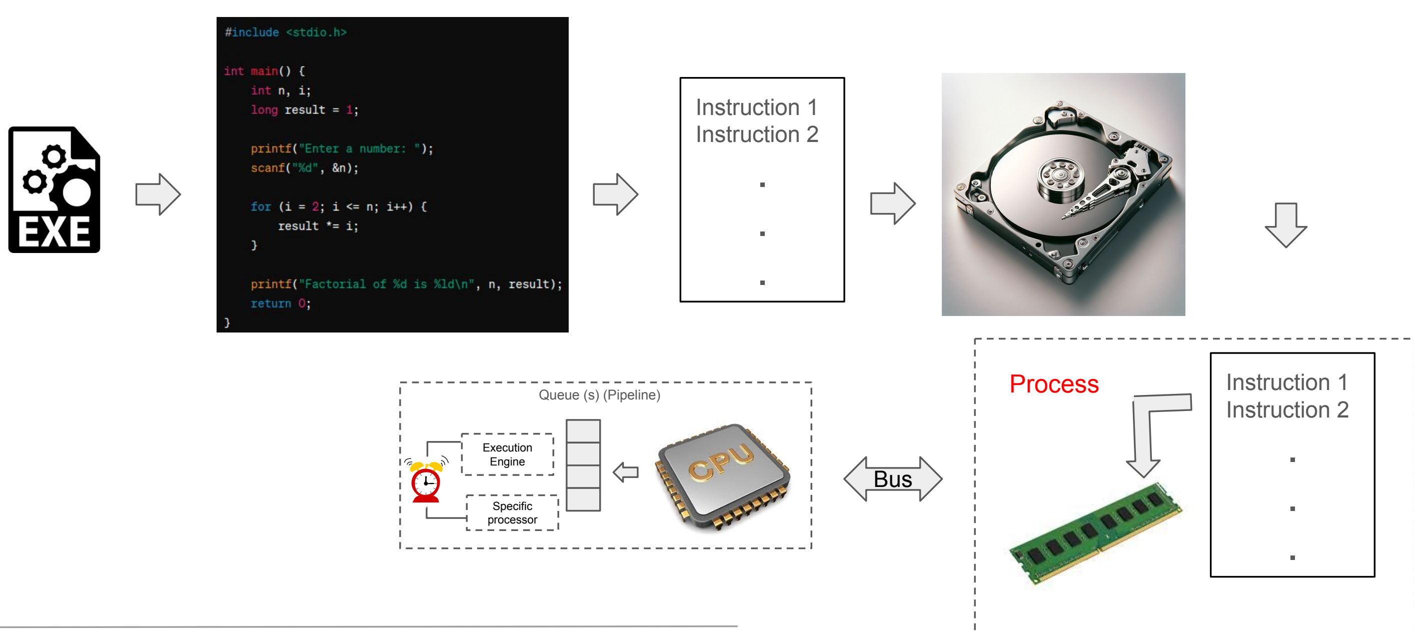
Start Time	End Time	Duration	SpanId	OperationName	TraceID	ThreadID	RUN	RUNSYSTEMCALL	INTERRUPTED	WAITCPU	WAITUNKNOWN	WAITFORK	UNKNOWN
11:06:56.933 952 078	11:06:56.933 980 155	28,077	4442450064020525663	*executor.ShowExec.Next	1618081973820362342	1805943	17108	12915	0	0	0	0	0
11:06:56.934 022 060	11:06:56.934 042 663	20,603	6023047575761396435	*executor.ShowExec.Next	1618081973820362342	1805943	7188	15709	0	0	0	0	0
11:07:46.711 230 210	11:07:46.711 249 416	19,206	2871436018761349651	*executor.ShowExec.Next	7968810417120810813	1813228	10054	11450	0	0	0	0	0
11:07:46.711 292 718	11:07:46.711 303 334	10,616	1574938190389480438	*executor.ShowExec.Next	7968810417120810813	1813228	4326	8727	0	0	0	0	0
11:06:56.933 723 277	11:06:56.933 843 056	119,779	7456353795818001242	executor.Compile	1618081973820362342	1805943	68216	53966	0	0	0	0	0
11:07:46.711 033 606	11:07:46.711 132 920	99,314	2159668538522373865	executor.Compile	7968810417120810813	1813228	52290	49358	0	0	0	0	0
11:06:56.933 567 600	11:06:56.933 592 324	24,724	3369336188777566051	Search: Duplicate	1618081973820362342	1805943	10539	16130	0	0	0	0	0
11:07:46.710 746 696	11:07:46.710 772 468	25,772	8310292673920133684	Search: Duplicate	7968810417120810813	1813228	10960	16687	0	0	0	0	0

We need more detail informations to answer this question!

What is happening inside the system?!

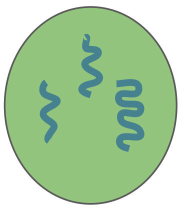
- How a program runs?
- What are the processes and threads?
- What are kernel and user space levels?
- What are process/ thread states?

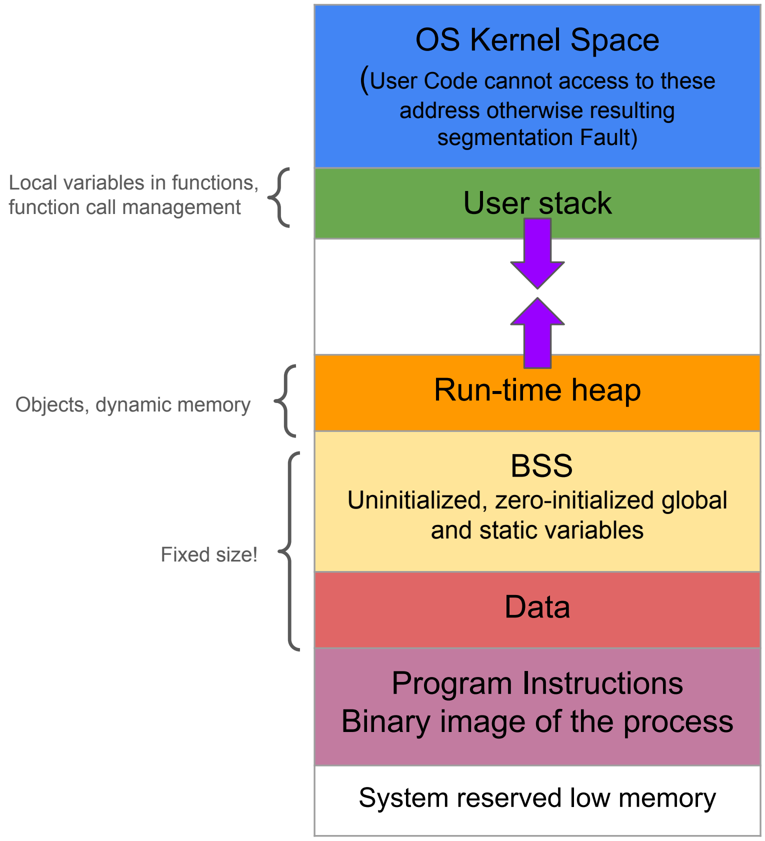
What is a process?



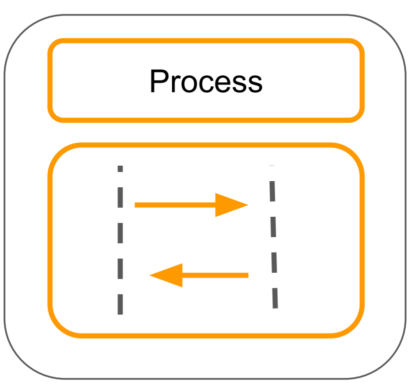
What is a process?

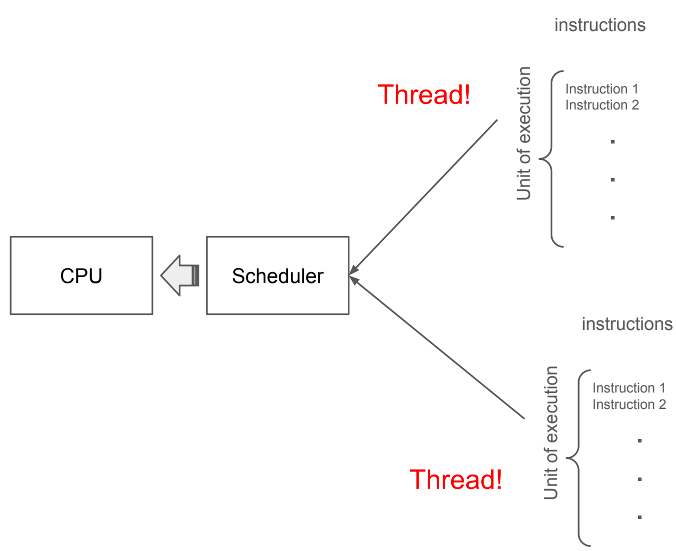
Program in execution!!





What is a thread?





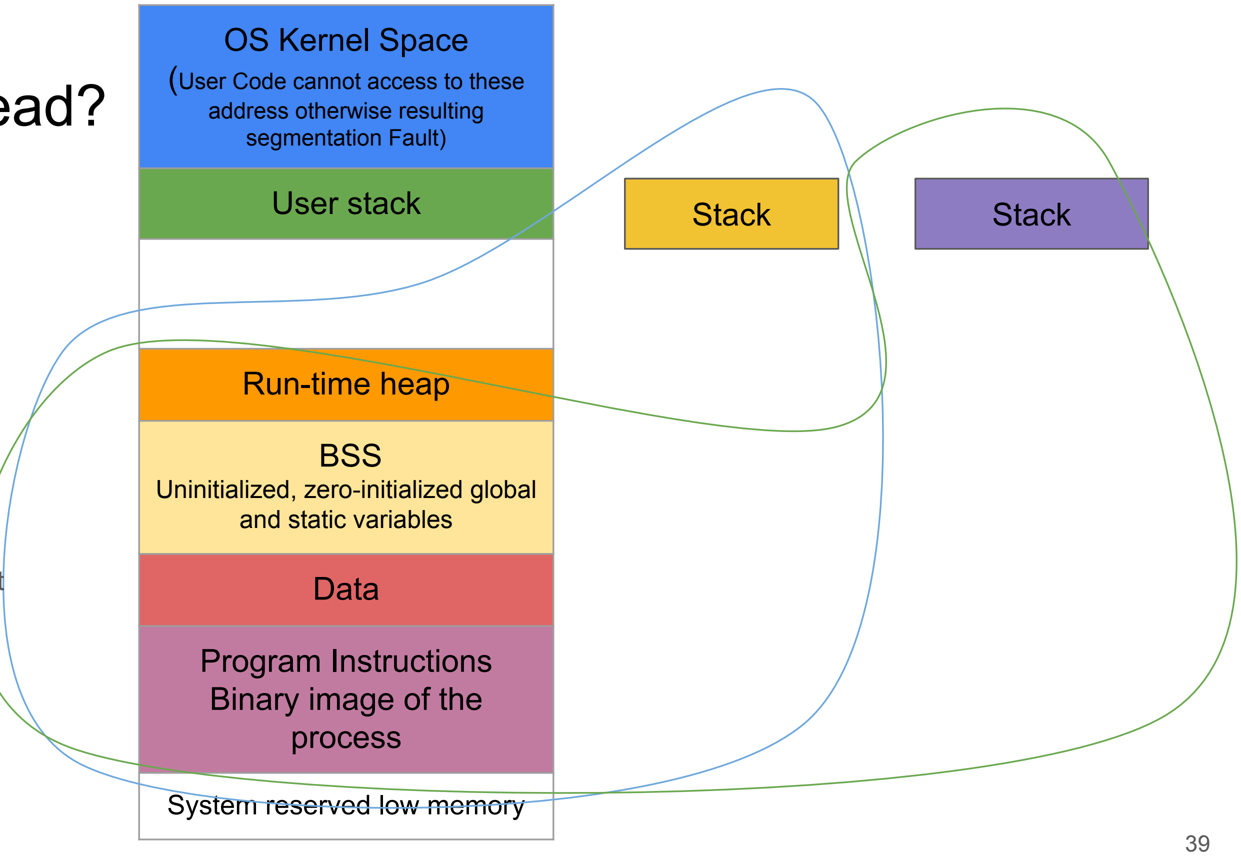
What is a thread?

A Thread contains

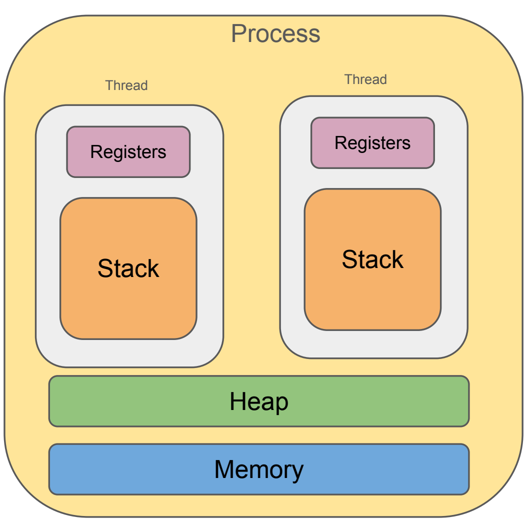
- A thread ID
- A program counter
- A register set
- A stack

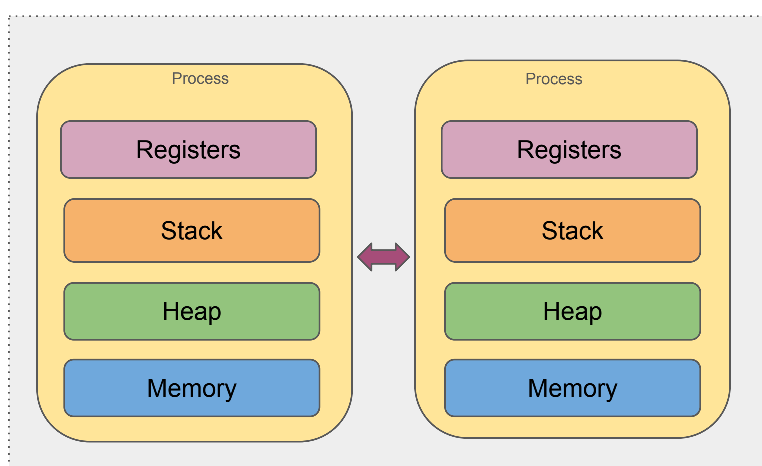
Benefits of threads

- Each process can do more than one task at a time
- Threads are lightweight however processes are heavyweight

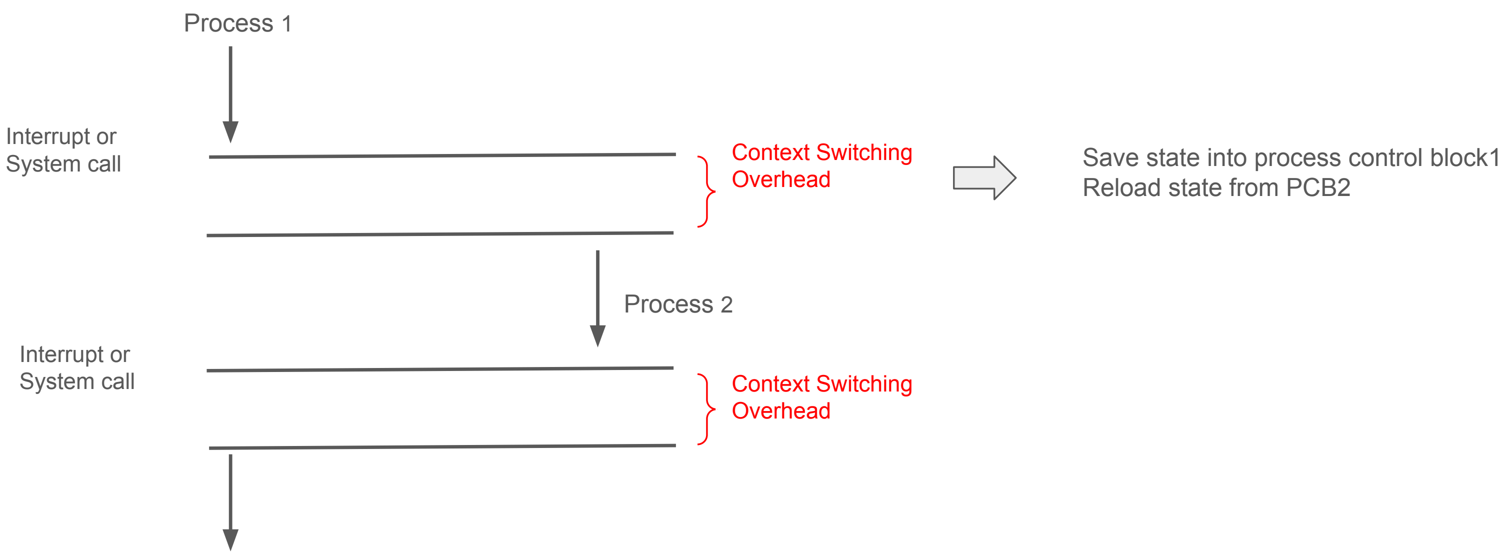


Differences between threads and Processes

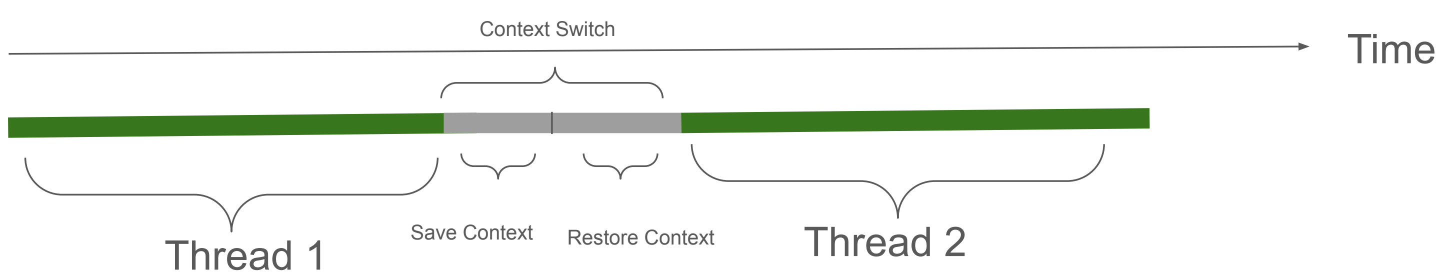




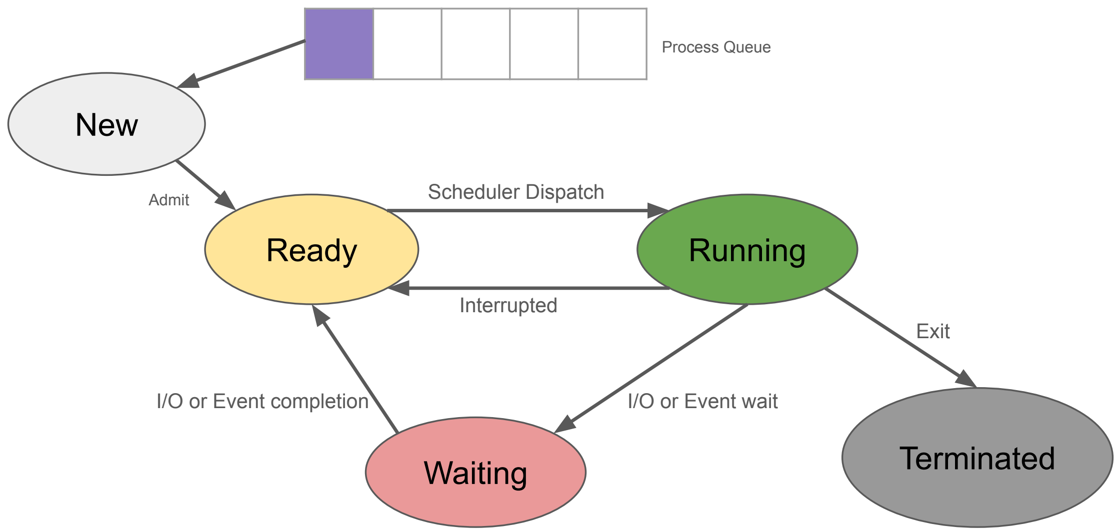
Context Switch in processes



Context Switch in threads

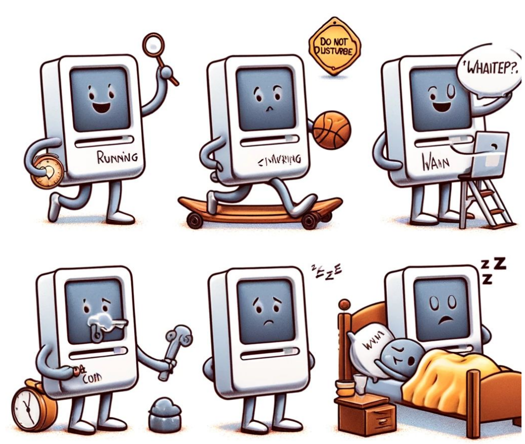


We have to know the process states

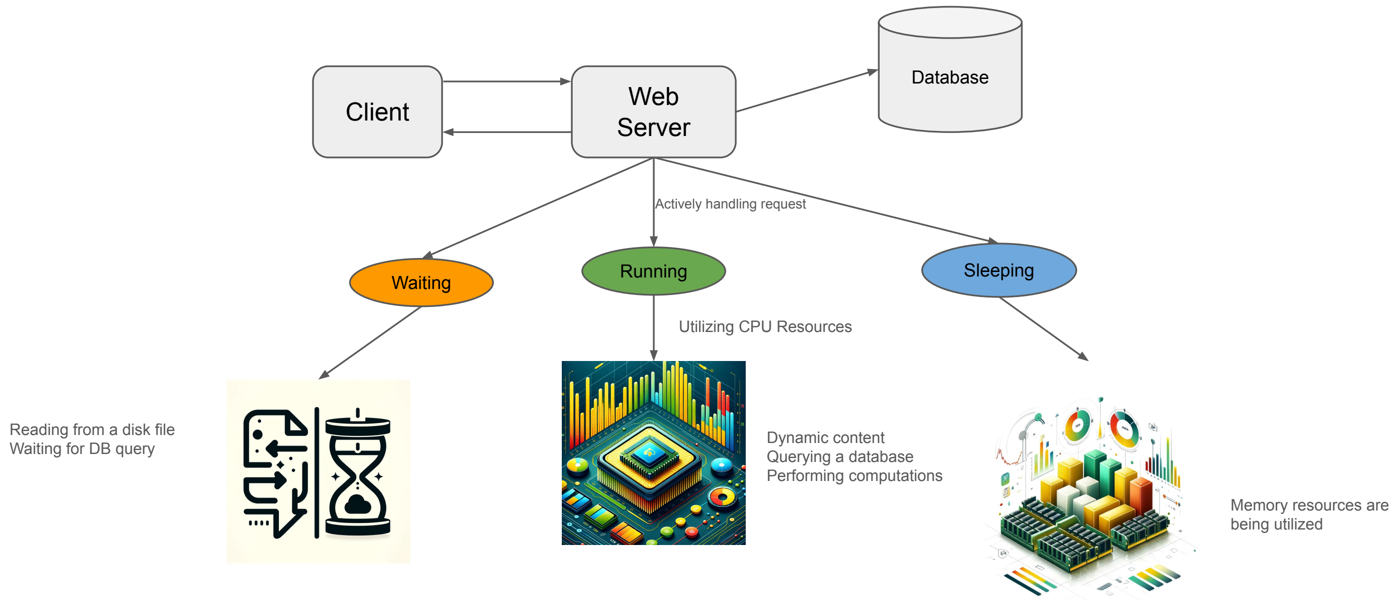


What the state of a process can tell us

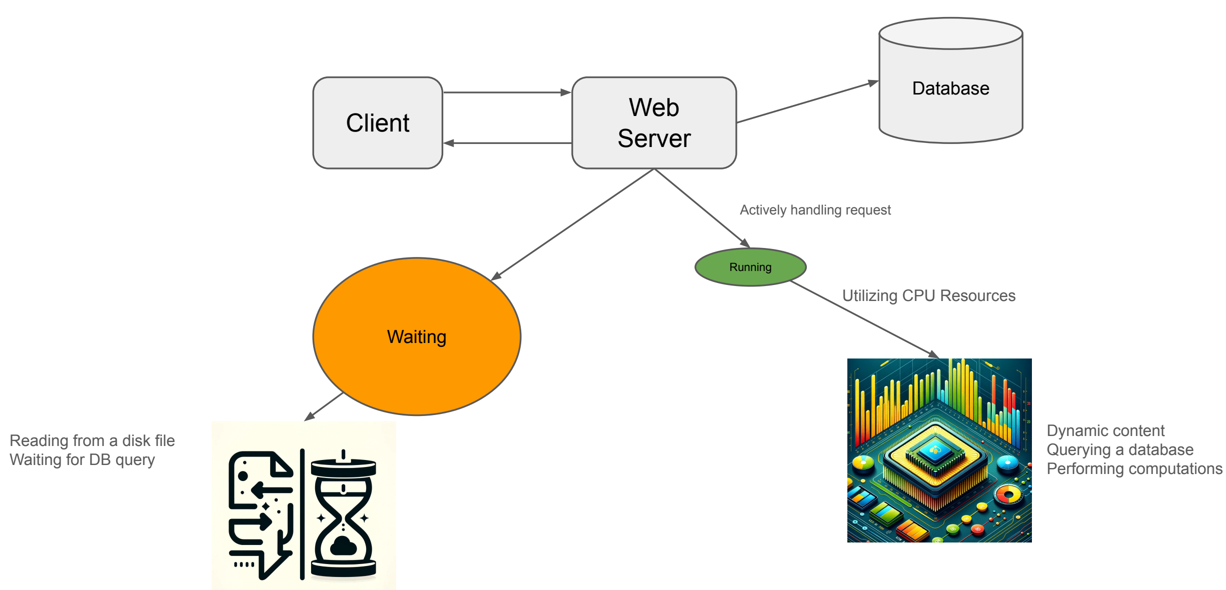
- Process Status
- Resource Utilization
- Prioritization
- I/O Status
- Inter-process Communication (IPC)
- Concurrency and Parallelism
- Life Cycle Management



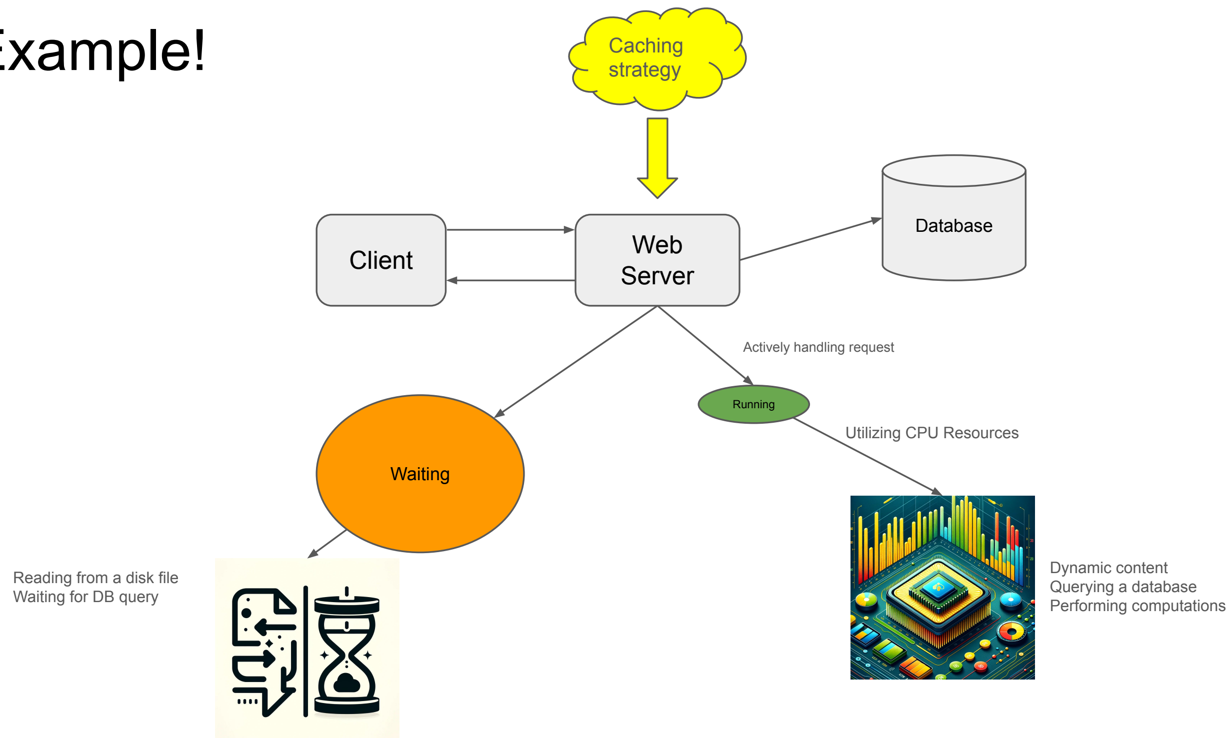
Example!



Example!

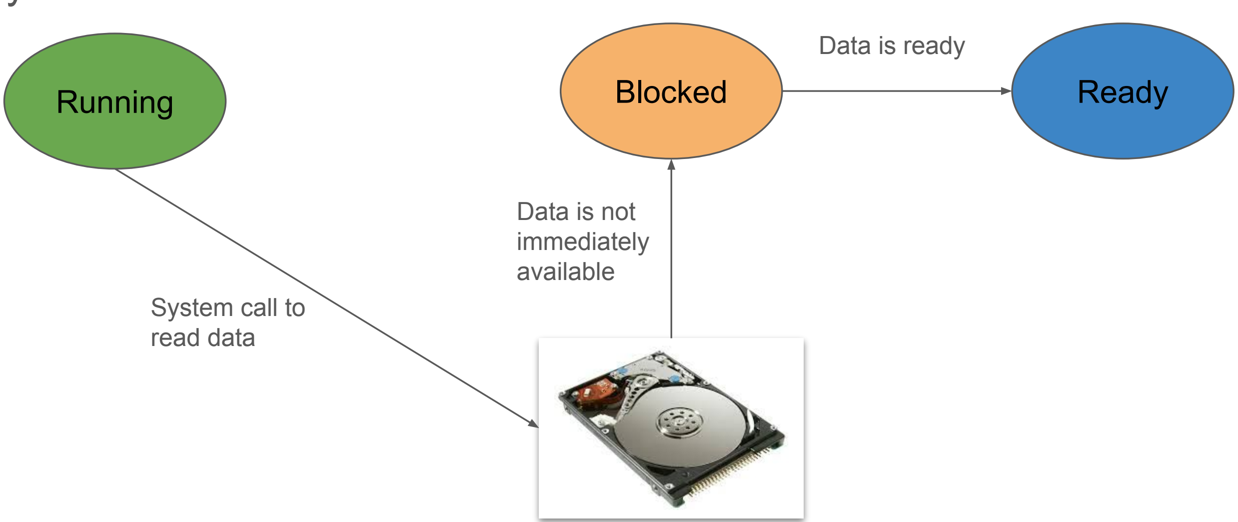


Example!



How we can know why the process is in the waiting state?

By system calls



What is Kernel?

Software

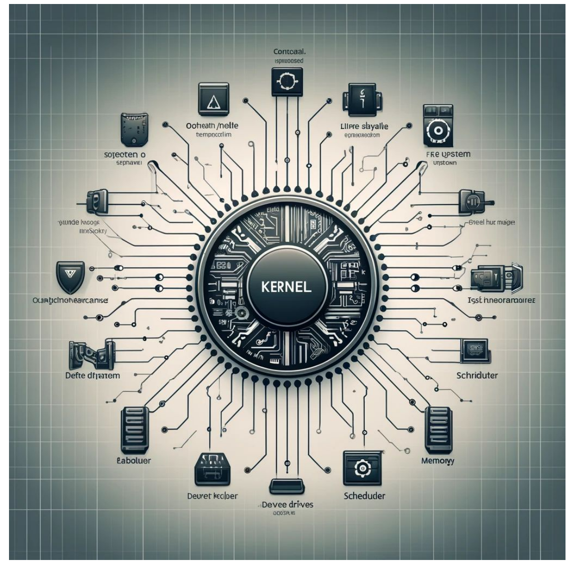
Kernel

Hardware

Software

Syscall Handler
File System
Device Drivers
Scheduler
Memory Manager

Hardware



What are system calls?

- Process Control
- File Manipulation
- Device Management
- Information Maintenance
- Communication

We have all the required information let's have an example!

What is the time complexity?

Complexity: Using Nested Loops	Complexity: Using a Hash Table

```
// Function to check for duplicates in an array using nested loops

bool hasDuplicates(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = i + 1; j < size; j++) {
            if (arr[i] == arr[j]) {
                return true; // Duplicate found
            }
        }
    }
    return false; // No duplicates
}
```

```
// Function to check for duplicates in an array using a simplistic hash table

bool hasDuplicatesOptimized(int arr[], int size) {
    bool hashTable[MAX_VALUE] = { false };

    for (int i = 0; i < size; i++) {
        if (arr[i] < 0 || arr[i] >= MAX_VALUE) {
            printf("Value out of range.\n");
            return false; // Value out of the assumed range
        }
        if (hashTable[arr[i]]) {
            return true; // Duplicate found
        }
        hashTable[arr[i]] = true;
    }
    return false; // No duplicates
}
```

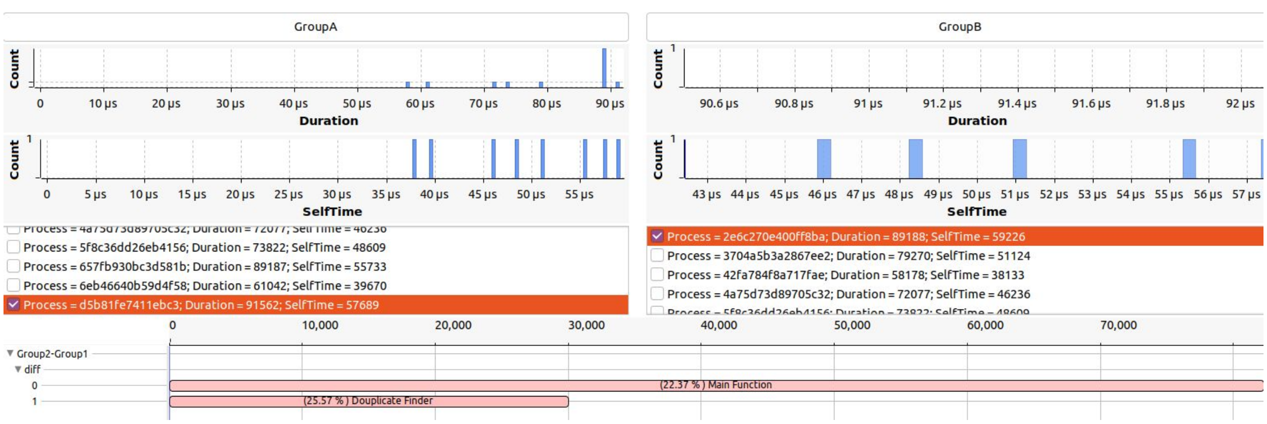
Actual Use case - C Language

O(n^2) Complexity: Using Nested Loops	O(n) Complexity: Using a Hash Table

```c
// Function to check for duplicates in an array using nested loops
// Time Complexity: O(n^2)
bool hasDuplicates(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = i + 1; j < size; j++) {
            if (arr[i] == arr[j]) {
                return true; // Duplicate found
            }
        }
    }
    return false; // No duplicates
}
```

```c
// Function to check for duplicates in an array using a simplistic hash table
// Time Complexity: O(n), with the caveat of known and limited range of input values
bool hasDuplicatesOptimized(int arr[], int size) {
    bool hashTable[MAX_VALUE] = { false };

    for (int i = 0; i < size; i++) {
        if (arr[i] < 0 || arr[i] >= MAX_VALUE) {
            printf("Value out of range.\n");
            return false; // Value out of the assumed range
        }
        if (hashTable[arr[i]]) {
            return true; // Duplicate found
        }
        hashTable[arr[i]] = true;
    }
    return false; // No duplicates
}
```


Differential Flame graph



Start Time	End Time	Duration	SpanId	OperationName	TraceID	ThreadID	WAITBI	RUN	RUNSYSTEMCALL
13:47:15.359 310 272	13:47:15.359 335 485	25,213	5776357360359181435	Douplicate Finder	6884938253811269974	162675	0	10890	16267
13:47:16.738 207 718	13:47:16.738 233 559	25,841	6989984953073631454	Douplicate Finder	5365431190380633138	162886	0	9004	19620
13:47:18.228 442 215	13:47:18.228 462 260	20,045	8646302322566141477	Douplicate Finder	4826302233701351342	163087	0	7677	14242
13:47:19.714 625 553	13:47:19.714 659 007	33,454	5233248868204268300	Douplicate Finder	7313767938840680475	163288	0	14800	21505
13:47:21.584 354 488	13:47:21.584 384 450	29,962	7523690523383491055	Douplicate Finder	3345091565387249850	163522	0	14730	18504
13:47:23.745 666 585	13:47:23.745 694 731	28,146	3868363194620979092	Douplicate Finder	3964475762937331426	163723	0	9423	21087
13:47:25.408 232 748	13:47:25.408 254 120	21,372	1666668865312156241	Douplicate Finder	7977113268090916696	163911	0	8515	14801
13:47:27.035 865 759	13:47:27.035 899 632	33,873	5156404004869947743	Douplicate Finder	962505875242740675	164118	0	11937	24717
13:47:15.359 290 786	13:47:15.359 364 608	73,822	6884938253811269974	Main Function	6884938253811269974	162675	0	29880	45871
13:47:16.738 189 070	13:47:16.738 261 147	72,077	5365431190380633138	Main Function	5365431190380633138	162886	0	25691	49222
13:47:18.228 425 593	13:47:18.228 483 771	58,178	4826302233701351342	Main Function	4826302233701351342	163087	0	22057	38399
13:47:19.714 602 785	13:47:19.714 691 972	89,187	7313767938840680475	Main Function	7313767938840680475	163288	0	38678	53344
13:47:21.584 331 231	13:47:21.584 420 419	89,188	3345091565387249850	Main Function	3345091565387249850	163522	0	39025	53488
13:47:23.745 646 959	13:47:23.745 726 229	79,270	3964475762937331426	Main Function	3964475762937331426	163723	0	31904	50133
13:47:25.408 216 405	13:47:25.408 277 447	61,042	7977113268090916696	Main Function	7977113268090916696	163911	0	24711	38259
13:47:27.035 842 432	13:47:27.035 933 994	91,562	962505875242740675	Main Function	962505875242740675	164118	0	37350	57536

What is Software Performance

- Software Aspects
  - Code efficiency
  - **Caching strategy**
  - Architecture and Design
  - Concurrency and Parallelism
- Hardware Aspects
  - Processor Speed Core Count
  - Memory (RAM)
  - Storage (HDD/SSD)
  - Network Speed
- Other Factors

Caching Strategy

No-caching

```go
import (
	"database/sql"
	"fmt"
	"time"

	_ "github.com/mattn/go-sqlite3"
)

// User struct to hold user details
type User struct {
	ID   int
	Name string
}

// Global cache variable
var userCache = make(map[int]User)

// Non-caching version
func getUserDetailsWithoutCache(userID int) (*User, error) {
	db, err := sql.Open("sqlite3", "users.db")
	if err != nil {
		return nil, err
	}
	defer db.Close()

	var user User
	err = db.QueryRow("SELECT id, name FROM users WHERE id = ?", userID).Scan(&user.ID, &user.Name)
	if err != nil {
		return nil, err
	}

	return &user, nil
}
```

Caching

```go
import (
	"database/sql"
	"fmt"
	"time"

	_ "github.com/mattn/go-sqlite3"
)

// User struct to hold user details
type User struct {
	ID   int
	Name string
}

// Global cache variable
var userCache = make(map[int]User)


// Caching version
func getUserDetailsWithCache(userID int) (*User, error) {
	if user, found := userCache[userID]; found {
		return &user, nil
	}

	user, err := getUserDetailsWithoutCache(userID)
	if err != nil {
		return nil, err
	}

	userCache[userID] = *user
	return user, nil
}
```

Execution time without cache: 0.05859804153442383 seconds
Execution time with cache: 0.0003237724304199219 seconds

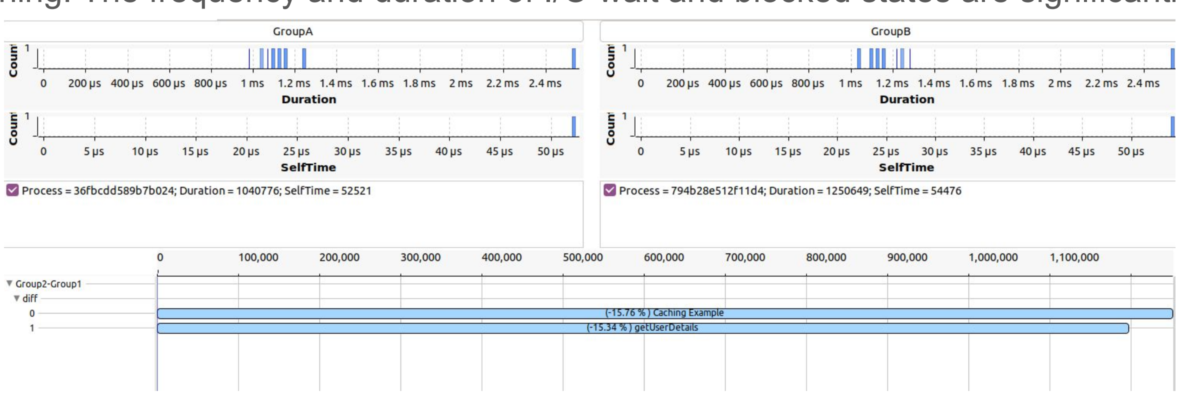
Expectation: Increase in blocked time and wait time since we need to fetch data from database
More context switch and cpu execution time for handling connection

What do you expect?

- Without Caching: Expect more frequent and longer periods in the I/O wait and possibly blocked states
- With Caching: The frequency and duration of I/O wait and blocked states are significantly reduced

Expectations are Confirmed!

- Without Caching: Expect more frequent and longer periods in the I/O wait and possibly blocked states
- With Caching: The frequency and duration of I/O wait and blocked states are significantly reduced



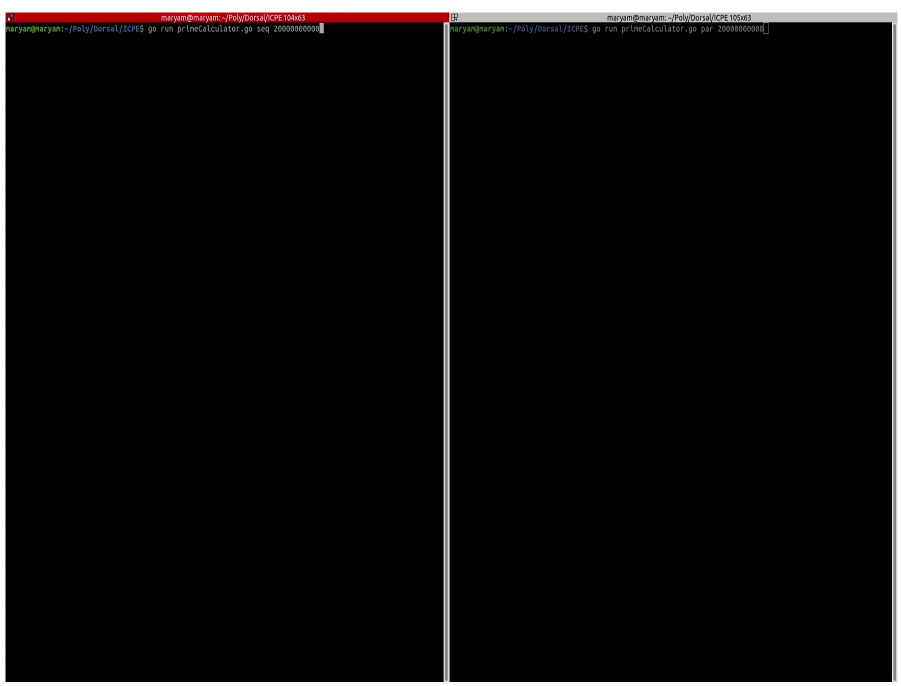
Start Time	End Time	Duration	SpanId	OperationName	TraceID	ThreadID	RUN	RUNSYSTEMCALL	WAITCPU	WAITUNKNOWN
14:33:19.065 559 991	14:33:19.066 219 644	659,653	878708228217677786	getUserDetailsWithoutCache	8698218330698166442	246961	449304	212619	0	0
14:33:19.066 232 914	14:33:19.066 631 430	398,516	9073351341883728441	getUserDetailsWithCache	8698218330698166442	246961	193475	207244	0	0
14:33:19.065 544 346	14:33:19.066 654 477	1,110,131	8698218330698166442	Caching Example	8698218330698166442	246961	666026	445762	0	0
14:33:20.892 361 205	14:33:20.893 006 121	644,916	7929464920158992492	getUserDetailsWithoutCache	7373370459793230796	247172	398048	233423	0	16272
14:33:20.893 021 137	14:33:20.893 499 970	478,833	7248847200101164513	getUserDetailsWithCache	7373370459793230796	247172	236359	219809	25909	0

What is Software Performance

- Software Aspects
  - Code efficiency
  - Caching strategy
  - Architecture and Design
  - **Concurrency and Parallelism**
- Hardware Aspects
  - Processor Speed Core Count
  - Memory (RAM)
  - Storage (HDD/SSD)
  - Network Speed
- Other Factors

Concurrency and Parallelism

CPU Bound Example:



Parallel and sequential

```go
// isPrime checks if a number is a prime
func isPrime(number int) bool {
    if number <= 1 {
        return false
    }
    for i := 2; i <= int(math.Floor(math.Sqrt(float64(number)))); i++ {
        if number%i == 0 {
            return false
        }
    }
    return true
}

// calculatePrimesSeq calculates the first n primes sequentially
func calculatePrimesSeq(upperLimit int) []int {
    primes := []int{}
    for num := 2; num <= upperLimit; num++ {
        if isPrime(num) {
            primes = append(primes, num)
            fmt.Printf("Found Sequential Prime: %d\n", num) // Print found primes
        }
    }
    return primes
}
```

```go
// isPrime checks if a number is a prime
func isPrime(number int) bool {
    if number <= 1 {
        return false
    }
    for i := 2; i <= int(math.Floor(math.Sqrt(float64(number)))); i++ {
        if number%i == 0 {
            return false
        }
    }
    return true
}

// calculatePrimesParallel calculates the first n primes in parallel
func calculatePrimesParallel(upperLimit int) []int {
    var primes []int
    var mu sync.Mutex
    var wg sync.WaitGroup

    // Use a channel to generate numbers to be checked for primality
    numbers := make(chan int, 100) // Buffered channel

    // Start a fixed number of worker goroutines to check numbers for primality in parallel
    for i := 0; i < 10; i++ { // Limit concurrency to prevent excessive resource usage
        wg.Add(1)
        go func() {
            defer wg.Done()
            for number := range numbers {
                if isPrime(number) {
                    mu.Lock()
                    primes = append(primes, number)
                    fmt.Printf("Found Parallel Prime: %d\n", number) // Print found primes
                    mu.Unlock()
                }
            }
        }()
    }
```


What do you expect?

Performance Metrics!

Start Time	End Time	Duration	SpanId	OperationName	TraceID	ThreadID	WAITBLOCKED	RUN	RUNSYSTEMCALL	WAITCPU
10:51:16.862 378 440	10:51:16.862 406 586	28,146	8409096035200089774	calculateFibunacciParallel	2881530242770011486	2410233	0	11378	19128	0
10:51:16.862 313 417	10:51:16.862 464 834	151,417	8717727913264033703	calculateFibunacciParallel	2881530242770011486	2410227	53358	35115	79117	9567
10:51:16.862 346 033	10:51:16.862 506 809	160,776	9169217551823845228	calculateFibunacciParallel	2881530242770011486	2410227	53358	28973	70736	9567
10:51:16.862 278 217	10:51:16.862 517 355	239,138	2881530242770011486	fibunachi	2881530242770011486	2410227	53358	66529	115698	9567
10:51:21.713 643 539	10:51:21.713 676 923	33,384	6514851310001073122	calculateFibunacciSequen	5476396147119877088	2410540	0	9424	25903	0
10:51:21.713 692 777	10:51:21.713 712 263	19,486	7465671310475914216	calculateFibunacciSequen	5476396147119877088	2410540	0	5235	16687	0
10:51:21.713 632 643	10:51:21.713 729 933	97,290	8807225039319203146	calculateFibunacciSequen	5476396147119877088	2410540	0	33155	66468	0
10:51:21.713 609 456	10:51:21.713 754 727	145,271	5476396147119877088	fibunachi	5476396147119877088	2410540	0	52978	94605	0

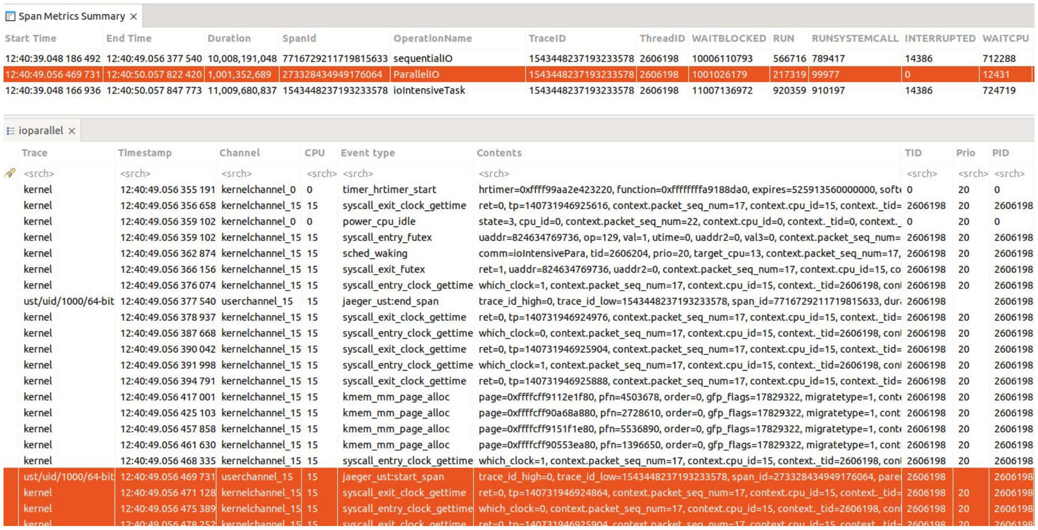
parallel ×

Trace	Timestamp	Channel	CPU	Event type	Contents	T
<srch>	<srch>	<srch>	<srch>	<srch>	<srch>	<
kernel	10:51:16.862 308 668	kernelchannel_5	5	sched_switch	prev_comm=swapper/5, prev_tid=0, prev_prio=20, prev_state=0, next_comm=fibunachi-par	0
kernel	10:51:16.862 309 576	kernelchannel_14	14	writeback_dirty_page	name=259:0, ino=21764999, index=2006, context.packet_seq_num=21, context.cpu_id=14, c	1
kernel	10:51:16.862 309 576	kernelchannel_8	8	power_cpu_idle	state=2, cpu_id=8, context.packet_seq_num=10, context.cpu_id=8, context._tid=0, context._	0
kernel	10:51:16.862 311 042	kernelchannel_14	14	writeback_mark_inode_dirty	name=259:0, ino=21764999, state=7, flags=4, context.packet_seq_num=21, context.cpu_id=	1
kernel	10:51:16.862 311 042	kernelchannel_4	4	syscall_exit_clock_gettime	ret=0, tp=140730733719008, context.packet_seq_num=8, context.cpu_id=4, context._tid=24	2
kernel	10:51:16.862 311 042	kernelchannel_5	5	syscall_exit_futex	ret=0, uaddr=824634769736, uaddr2=0, context.packet_seq_num=10, context.cpu_id=5, con	2
kernel	10:51:16.862 312 439	kernelchannel_1	1	syscall_entry_recvmsg	fd=33, msg=140722553225216, flags=0, context.packet_seq_num=7, context.cpu_id=1, cont	1
kernel	10:51:16.862 312 439	kernelchannel_14	14	writeback_mark_inode_dirty	name=259:0, ino=21764999, state=7, flags=7, context.packet_seq_num=21, context.cpu_id=	1
kernel	10:51:16.862 312 439	kernelchannel_4	4	syscall_entry_clock_gettime	which_clock=1, context.packet_seq_num=8, context.cpu_id=4, context._tid=2410227, contex	2
kernel	10:51:16.862 312 439	kernelchannel_5	5	syscall_entry_clock_gettime	which_clock=1, context.packet_seq_num=10, context.cpu_id=5, context._tid=2410233, conte	2
kernel	10:51:16.862 313 417	kernelchannel_14	14	writeback_dirty_inode_start	name=259:0, ino=21764999, state=7, flags=7, context.packet_seq_num=21, context.cpu_id=	1
ust/uid/1000/64-bit	10:51:16.862 313 417	userchannel_4	4	jaeger_ust:start_span	trace_id_high=0, trace_id_low=2881530242770011486, span_id=8717727913264033703, par	2
kernel	10:51:16.862 314 814	kernelchannel_1	1	kmem_kfree	call_site=0xffffffffa9aaa2cd, ptr=0x0, context.packet_seq_num=7, context.cpu_id=1, contex	1
kernel	10:51:16.862 314 814	kernelchannel_14	14	kmem_cache_alloc	call_site=0xffffffffa94be865, ptr=0xffff99a4000c5a80, bytes_req=56, bytes_alloc=56, gfp_fla	1
kernel	10:51:16.862 314 814	kernelchannel_4	4	syscall_exit_clock_gettime	ret=0, tp=140730733717984, context.packet_seq_num=8, context.cpu_id=4, context._tid=24	2
kernel	10:51:16.862 314 814	kernelchannel_5	5	syscall_exit_clock_gettime	ret=0, tp=139832860861152, context.packet_seq_num=10, context.cpu_id=5, context._tid=2	2
kernel	10:51:16.862 315 792	kernelchannel_1	1	syscall_exit_recvmsg	ret=-11, msg=140722553225216, context.packet_seq_num=7, context.cpu_id=1, context._tid	1
kernel	10:51:16.862 315 792	kernelchannel_14	14	block_touch_buffer	dev=271581186, sector=87032280, size=4096, context.packet_seq_num=21, context.cpu_id=	1
kernel	10:51:16.862 315 792	kernelchannel_5	5	syscall_entry_clock_gettime	which_clock=1, context.packet_seq_num=10, context.cpu_id=5, context._tid=2410233, conte	2
kernel	10:51:16.862 317 188	kernelchannel_1	1	syscall_entry_recvmsg	fd=33, msg=140722553225232, flags=0, context.packet_seq_num=7, context.cpu_id=1, cont	1
kernel	10:51:16.862 317 188	kernelchannel_14	14	kmem_cache_free	call_site=0xffffffffa94bf0e6, ptr=0xffff99a4000c5a80, name=mbcache, context.packet_seq_	1
kernel	10:51:16.862 318 166	kernelchannel_1	1	kmem_kfree	call_site=0xffffffffa9aaa2cd, ptr=0x0, context.packet_seq_num=7, context.cpu_id=1, contex	1
kernel	10:51:16.862 318 166	kernelchannel_14	14	writeback_dirty_inode	name=259:0, ino=21764999, state=7, flags=7, context.packet_seq_num=21, context.cpu_id=	1

Is parallel mechanism good for all situations?

- When Parallelism Is Beneficial
  - I/O-Intensive Tasks
  - Highly Concurrent Applications
  - Tasks with High Latency Operations
- When Parallelism May Not Be Beneficial
  - Computationally Intensive Tasks
  - Memory Bandwidth Saturation
  - Scaling Issues

I/O intensive tasks!

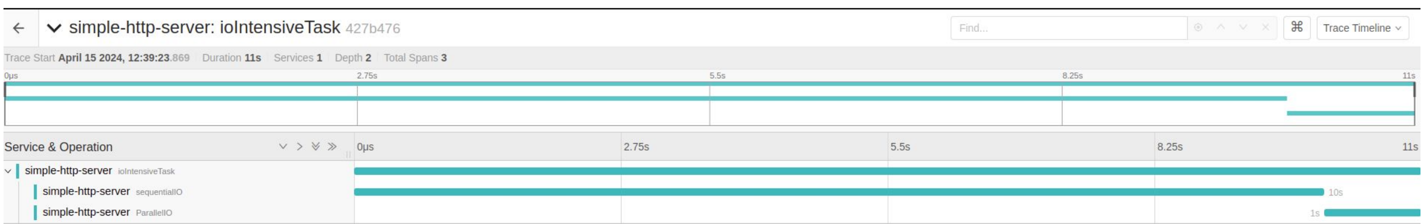


Span Metrics Summary

Start Time	End Time	Duration	SpanId	OperationName	TraceID	ThreadID	WAITBLOCKED	RUN	RUNSYSTEMCALL	INTERRUPTED	WAITCPU
12:40:39.048 186 492	12:40:49.056 377 540	10,008,191,048	7716729211719815633	sequentialIO	1543448237193233578	2606198	10006110793	566716	789417	14386	712288
12:40:49.056 469 731	12:40:50.057 822 420	1,001,352,689	273328434949176064	ParallelIO	1543448237193233578	2606198	1001026179	217319	99977	0	12431
12:40:39.048 166 936	12:40:50.057 847 773	11,009,680,837	1543448237193233578	ioIntensiveTask	1543448237193233578	2606198	11007136972	920359	910197	14386	724719

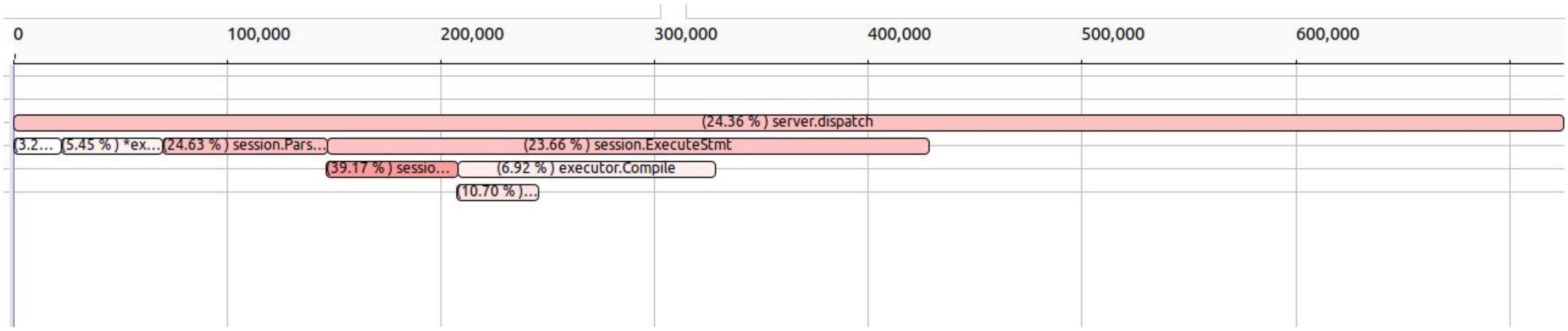
ioparallel

Trace	Timestamp	Channel	CPU	Event type	Contents	TID	Prio	PID
<srch>	<srch>	<srch>	<srch>	<srch>	<srch>	<srch>	<srch>	<srch>
kernel	12:40:49.056 355 191	kernelchannel_0	0	timer_hrtimer_start	hrtimer=0xffff99aa2e423220, function=0xffffffffa9188da0, expires=525913560000000, softe	0	20	0
kernel	12:40:49.056 356 658	kernelchannel_15	15	syscall_exit_clock_gettime	ret=0, tp=140731946925616, context.packet_seq_num=17, context.cpu_id=15, context._tid=	2606198	20	2606198
kernel	12:40:49.056 359 102	kernelchannel_0	0	power_cpu_idle	state=3, cpu_id=0, context.packet_seq_num=22, context.cpu_id=0, context._tid=0, context._	0	20	0
kernel	12:40:49.056 359 102	kernelchannel_15	15	syscall_entry_futex	uaddr=824634769736, op=129, val=1, utime=0, uaddr2=0, val3=0, context.packet_seq_num=	2606198	20	2606198
kernel	12:40:49.056 362 874	kernelchannel_15	15	sched_waking	comm=ioIntensivePara, tid=2606204, prio=20, target_cpu=13, context.packet_seq_num=17,	2606198	20	2606198
kernel	12:40:49.056 366 156	kernelchannel_15	15	syscall_exit_futex	ret=1, uaddr=824634769736, uaddr2=0, context.packet_seq_num=17, context.cpu_id=15, co	2606198	20	2606198
kernel	12:40:49.056 376 074	kernelchannel_15	15	syscall_entry_clock_gettime	which_clock=1, context.packet_seq_num=17, context.cpu_id=15, context._tid=2606198, con	2606198	20	2606198
ust/uid/1000/64-bit	12:40:49.056 377 540	userchannel_15	15	jaeger_ust:end_span	trace_id_high=0, trace_id_low=1543448237193233578, span_id=7716729211719815633, dur	2606198		2606198
kernel	12:40:49.056 378 937	kernelchannel_15	15	syscall_exit_clock_gettime	ret=0, tp=140731946924976, context.packet_seq_num=17, context.cpu_id=15, context._tid=	2606198	20	2606198
kernel	12:40:49.056 387 668	kernelchannel_15	15	syscall_entry_clock_gettime	which_clock=0, context.packet_seq_num=17, context.cpu_id=15, context._tid=2606198, con	2606198	20	2606198
kernel	12:40:49.056 390 042	kernelchannel_15	15	syscall_exit_clock_gettime	ret=0, tp=140731946925904, context.packet_seq_num=17, context.cpu_id=15, context._tid=	2606198	20	2606198
kernel	12:40:49.056 391 998	kernelchannel_15	15	syscall_entry_clock_gettime	which_clock=1, context.packet_seq_num=17, context.cpu_id=15, context._tid=2606198, con	2606198	20	2606198
kernel	12:40:49.056 394 791	kernelchannel_15	15	syscall_exit_clock_gettime	ret=0, tp=140731946925888, context.packet_seq_num=17, context.cpu_id=15, context._tid=	2606198	20	2606198
kernel	12:40:49.056 417 001	kernelchannel_15	15	kmem_mm_page_alloc	page=0xffffcff9112e1f80, pfn=4503678, order=0, gfp_flags=17829322, migratetype=1, conte	2606198	20	2606198
kernel	12:40:49.056 425 103	kernelchannel_15	15	kmem_mm_page_alloc	page=0xffffcff90a68a880, pfn=2728610, order=0, gfp_flags=17829322, migratetype=1, cont	2606198	20	2606198
kernel	12:40:49.056 457 858	kernelchannel_15	15	kmem_mm_page_alloc	page=0xffffcff9151f1e80, pfn=5536890, order=0, gfp_flags=17829322, migratetype=1, conte	2606198	20	2606198
kernel	12:40:49.056 461 630	kernelchannel_15	15	kmem_mm_page_alloc	page=0xffffcff90553ea80, pfn=1396650, order=0, gfp_flags=17829322, migratetype=1, cont	2606198	20	2606198
kernel	12:40:49.056 468 335	kernelchannel_15	15	syscall_entry_clock_gettime	which_clock=1, context.packet_seq_num=17, context.cpu_id=15, context._tid=2606198, con	2606198	20	2606198
ust/uid/1000/64-bit	12:40:49.056 469 731	userchannel_15	15	jaeger_ust:start_span	trace_id_high=0, trace_id_low=1543448237193233578, span_id=273328434949176064, pare	2606198		2606198
kernel	12:40:49.056 471 128	kernelchannel_15	15	syscall_exit_clock_gettime	ret=0, tp=140731946924864, context.packet_seq_num=17, context.cpu_id=15, context._tid=	2606198	20	2606198
kernel	12:40:49.056 475 389	kernelchannel_15	15	syscall_entry_clock_gettime	which_clock=0, context.packet_seq_num=17, context.cpu_id=15, context._tid=2606198, con	2606198	20	2606198
kernel	12:40:49.056 478 252	kernelchannel_15	15	syscall_exit_clock_gettime	ret=0, tp=140731946925904, context.packet_seq_num=17, context.cpu_id=15, context._tid=	2606198	20	2606198



What we created!

Differential Flame Graph and System metrics



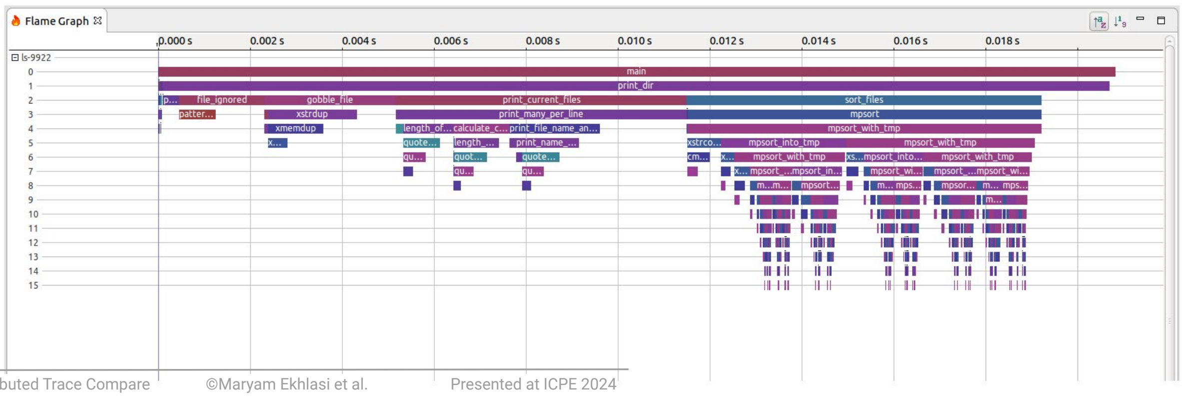
Start Time	End Time	Duration	SpanId	OperationName	TraceID	ThreadID	RUN	RUNSYSTEMCALL	INTERRUPTED	WAITCPU	WAITUNKNOWN	WAITFORK	UNKNOWN
11:06:56.933 952 078	11:06:56.933 980 155	28,077	4442450064020525663	*executor.ShowExec.Next	1618081973820362342	1805943	17108	12915	0	0	0	0	0
11:06:56.934 022 060	11:06:56.934 042 663	20,603	6023047575761396435	*executor.ShowExec.Next	1618081973820362342	1805943	7188	15709	0	0	0	0	0
11:07:46.711 230 210	11:07:46.711 249 416	19,206	2871436018761349651	*executor.ShowExec.Next	7968810417120810813	1813228	10054	11450	0	0	0	0	0
11:07:46.711 292 718	11:07:46.711 303 334	10,616	1574938190389480438	*executor.ShowExec.Next	7968810417120810813	1813228	4326	8727	0	0	0	0	0
11:06:56.933 723 277	11:06:56.933 843 056	119,779	7456353795818001242	executor.Compile	1618081973820362342	1805943	68216	53966	0	0	0	0	0
11:07:46.711 033 606	11:07:46.711 132 920	99,314	2159668538522373865	executor.Compile	7968810417120810813	1813228	52290	49358	0	0	0	0	0
11:06:56.933 567 600	11:06:56.933 592 324	24,724	3369336188777566051	Search: Duplicate	1618081973820362342	1805943	10539	16130	0	0	0	0	0
11:07:46.710 746 696	11:07:46.710 772 468	25,772	8310292673920133684	Search: Duplicate	7968810417120810813	1813228	10960	16687	0	0	0	0	0

Why differential flame graph

- Software performance analysis
- Latency detection
- New Software version or system updates
- Runtime discrepancies
- Pinpoint root causes

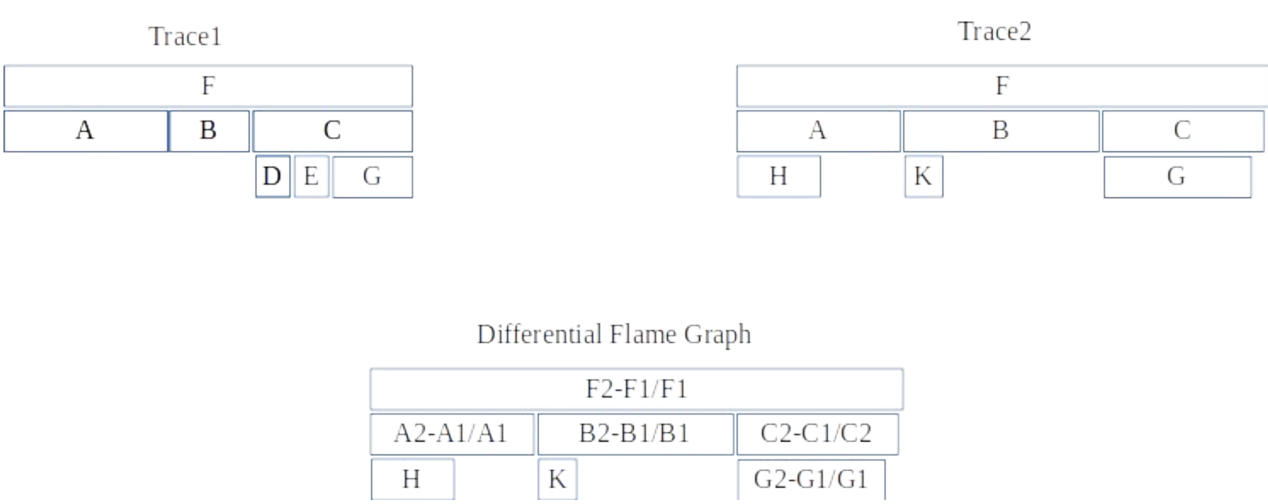
Flame Graph

- An aggregated view of the function calls from the call stack
- Each entry (box) represents a function in the stack.
- The x-axis represents total duration (execution time) and not absolute time.
- The width of an entry is the total time spent in that function, including time spent calling the children.
- Comprehensive yet simple and concise view, ideal for comparison



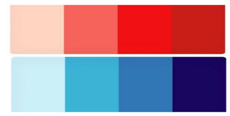
Differential Flame Graph-Computation

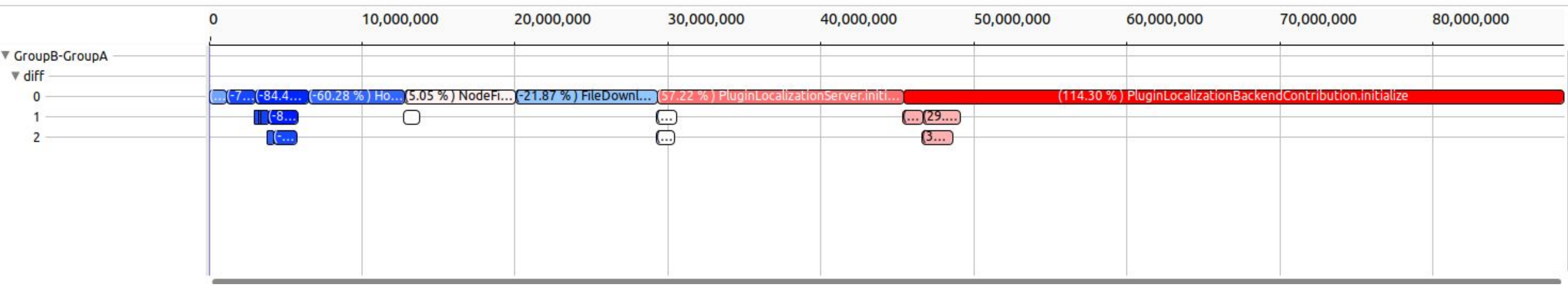
Suppose having two traces, we need to compare them



Differential Flame Graph-Representation

- Color
  - Flame Graph: The color of each function is randomly assigned based on the function name
  - Differential Flame Graph: colors are assigned based on the difference ratio
    - Red color palette for longer execution
    - Blue color palette for shorter execution
    - White for difference ratio between -5% and 5%



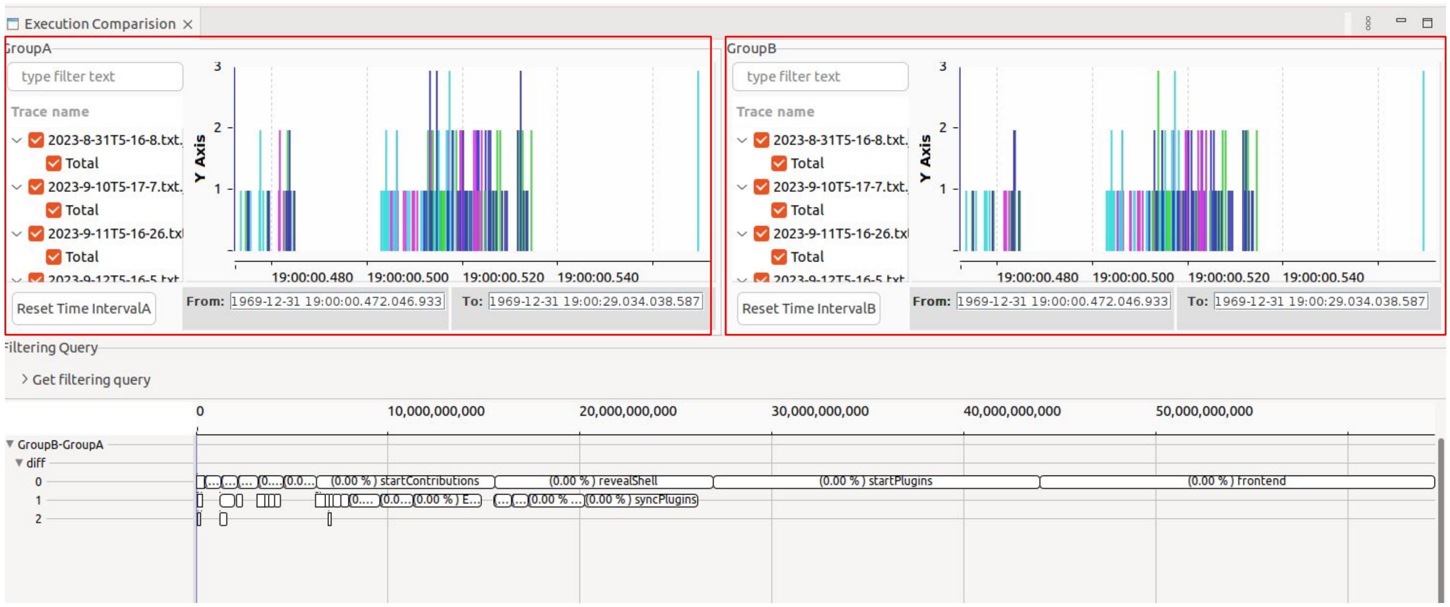


Execution Comparison

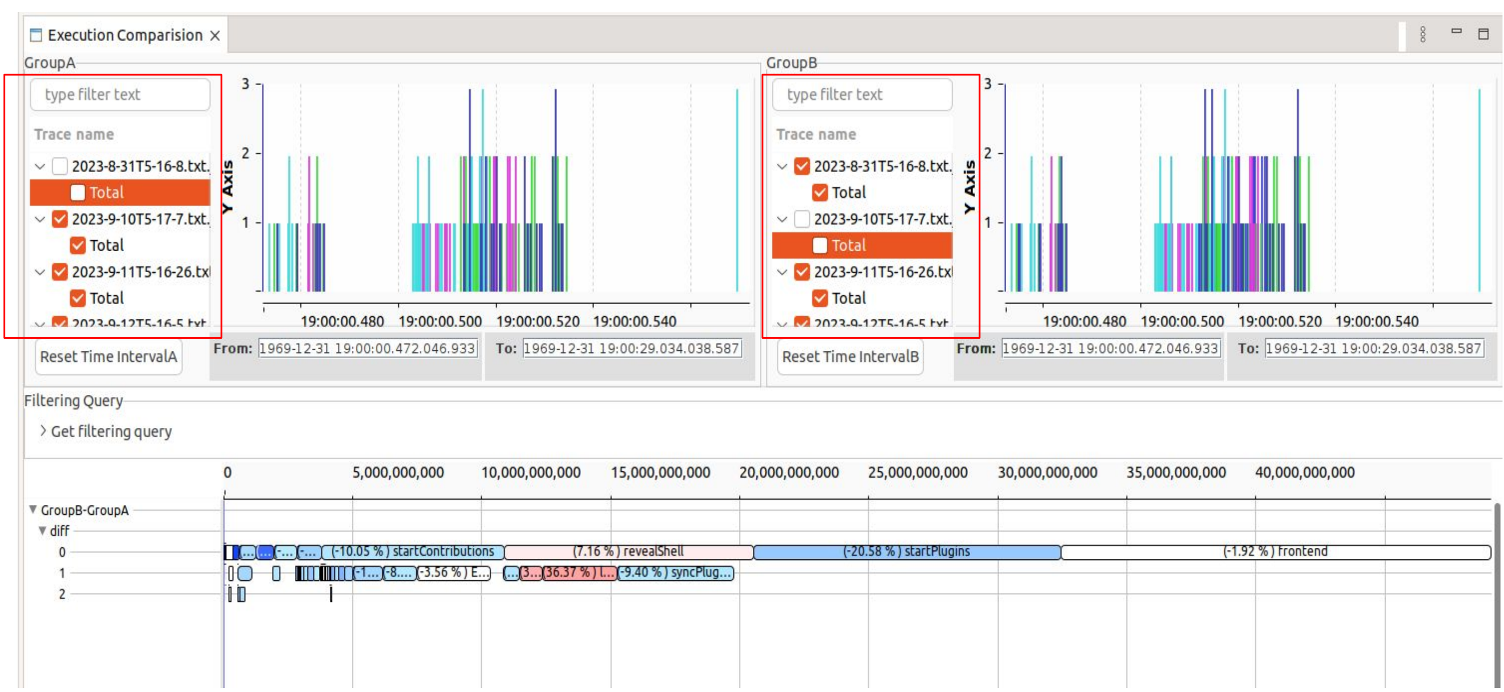
- We have two mode for execution comparison
  - Sequential traces
  - Distributed microservice-based system’s traces (request comparison)

Sequential trace comparison

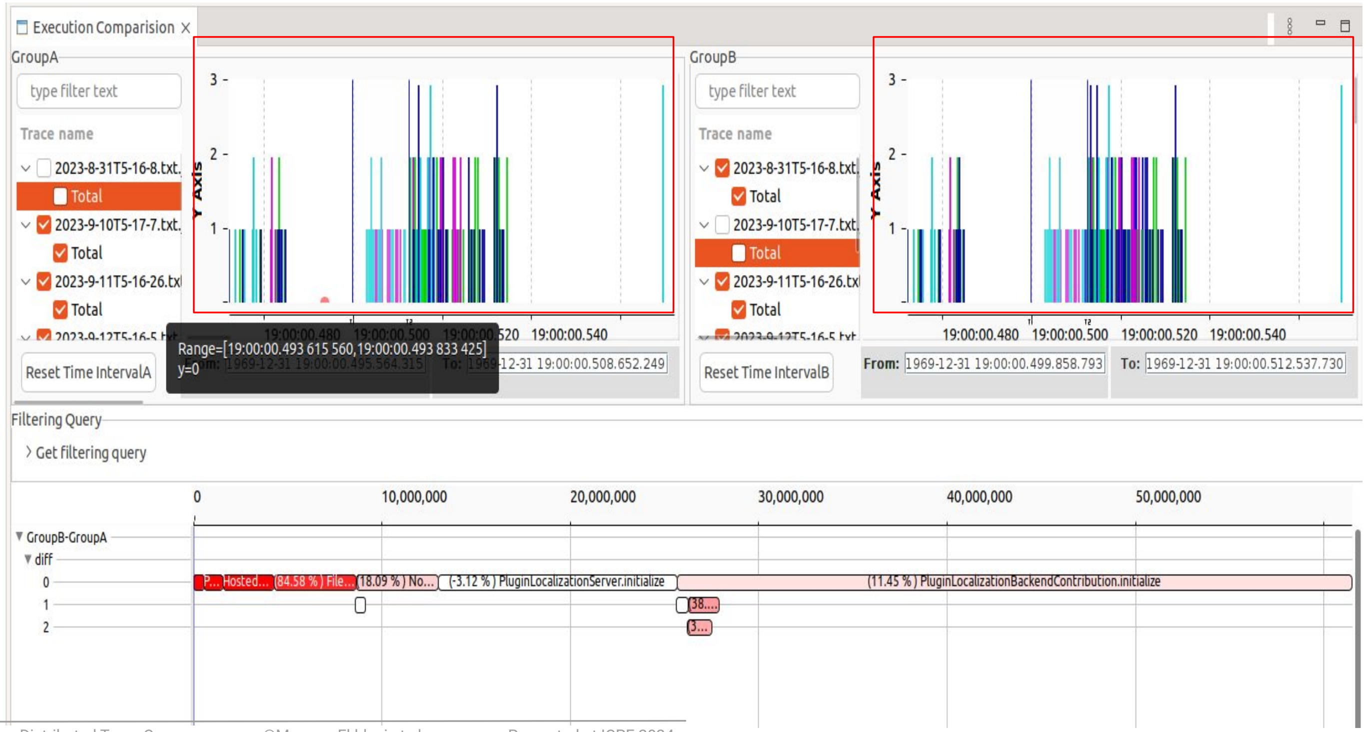
Select two groups to be compared



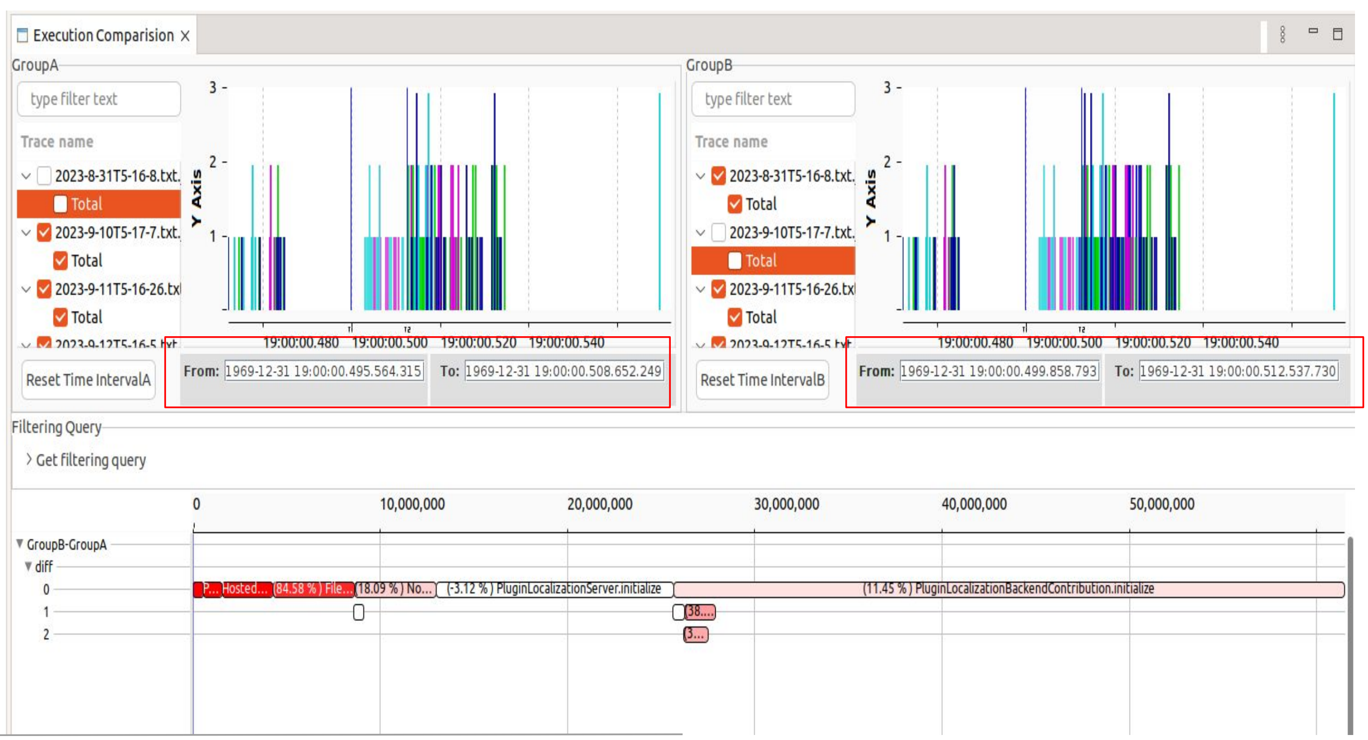
Select Relevant Traces



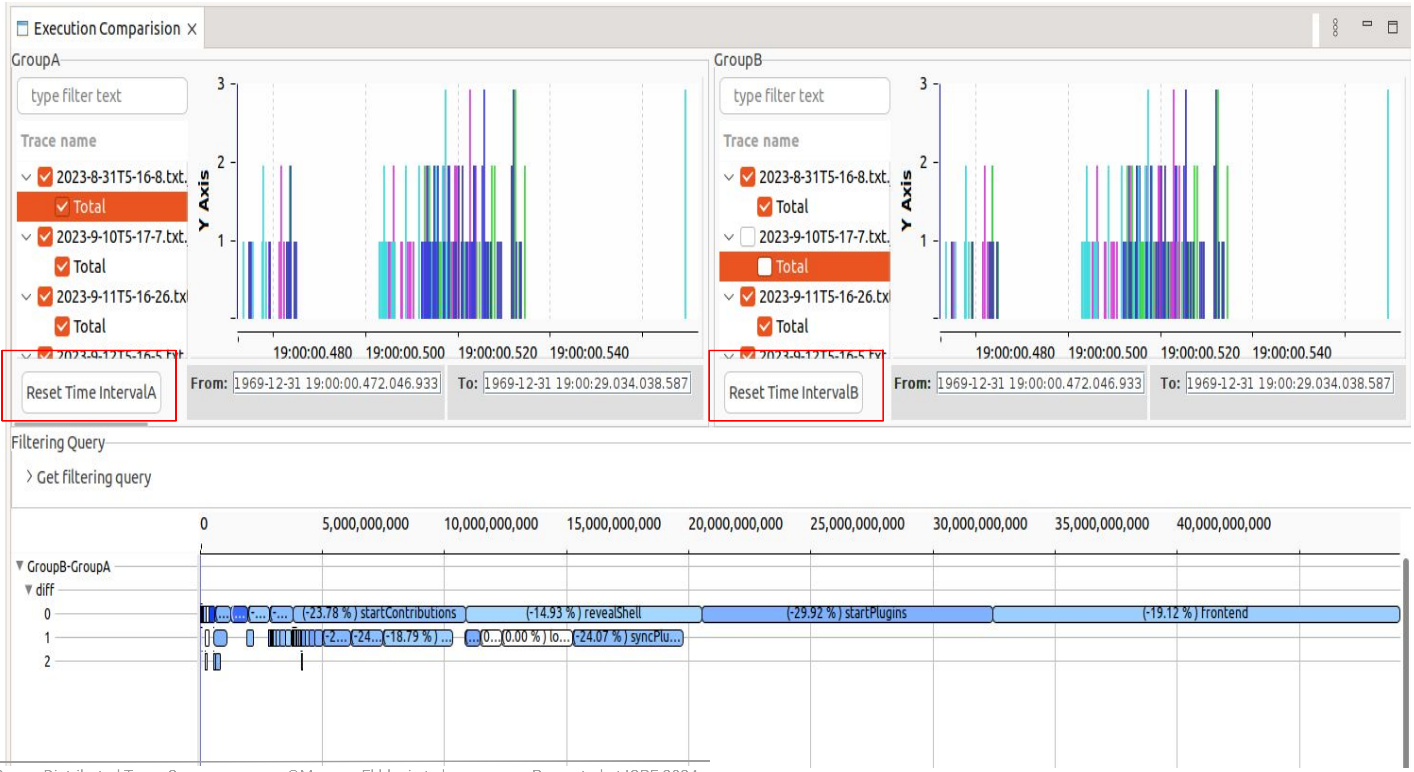
Select desired time range graphically



Select desired time range textually

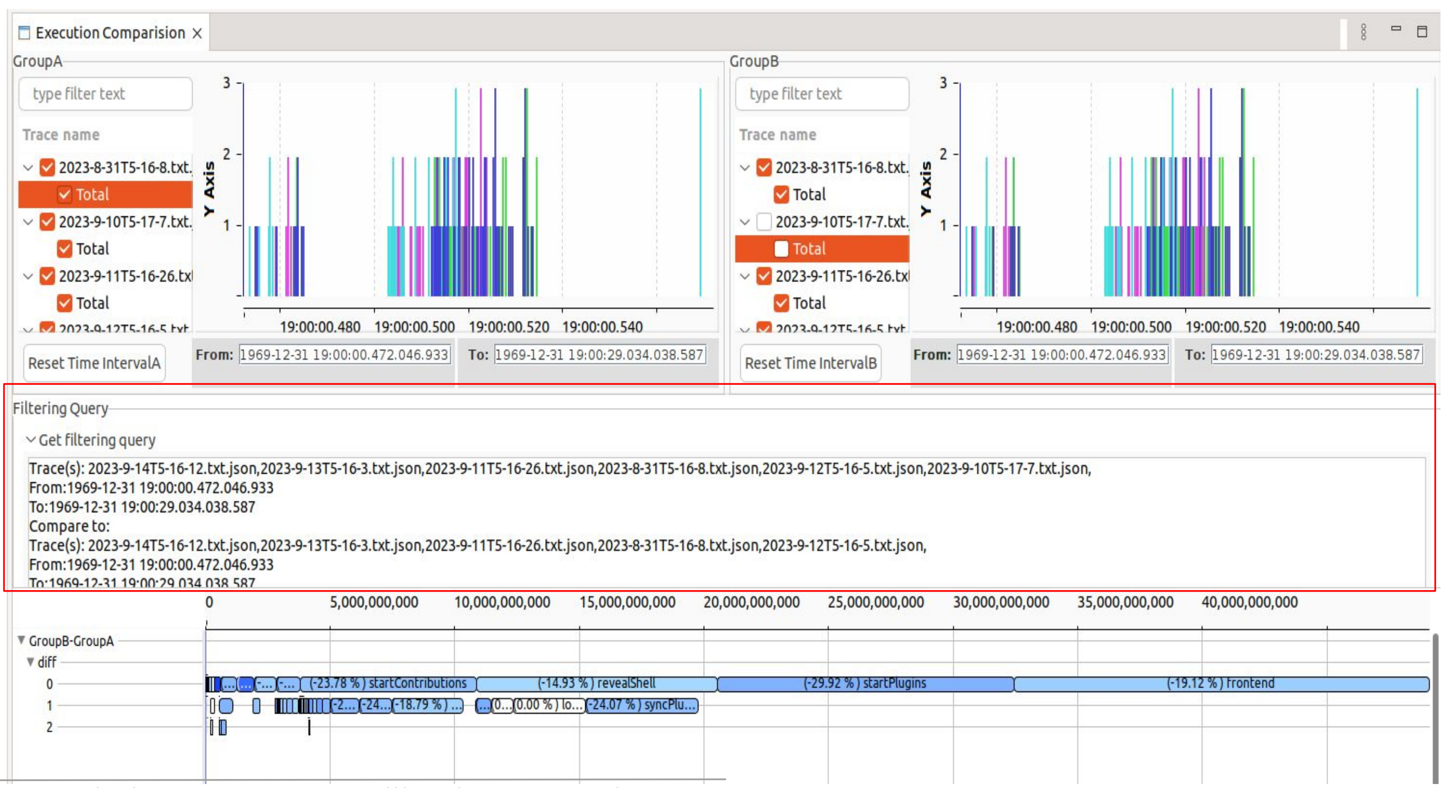


Reset to initial state



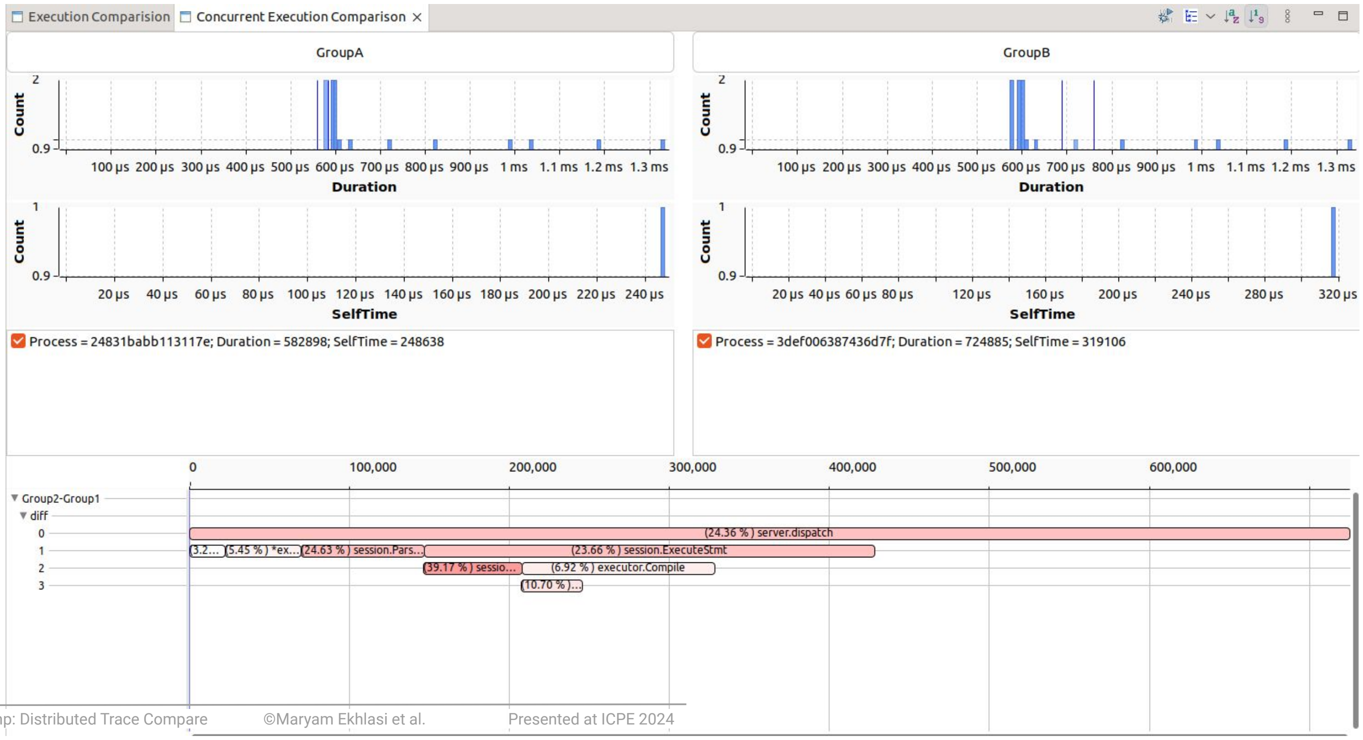
77

Experiment reproducibility



78

Request comparison



Let’s collect traces and see the result in TraceCompass!

Find all the commands here

https://github.com/maryamekhlasi/ICPE2024

Installing LTTng on Ubuntu

1. Add the LTTng Stable 2.13 PPA repository and update the list of packages

```
# apt-add-repository ppa:lttng/stable-2.13
# apt-get update
```

2. Install the main LTTng 2.13 packages

```
# apt-get install lttng-tools
# apt-get install lttng-modules-dkms
# apt-get install liblttng-ust-dev
```

Recording Kernel Events With LTTng

1. Checking the status of session daemon

```
$ systemctl status lttng-sessiond
```

2. Starting the session

```
$ sudo systemctl start lttng-sessiond
```

3. Create tracing session

```
$ sudo lttng create ICPE2024 --output="/home/maryam/Poly/Dorsal/traces/dataset/ICPE202"
```

4. Enable Events

```
$ sudo lttng enable-event -k -a
```

5. Start Tracing

```
$ sudo lttng start
```

6. Do something! Open a browser!

```
$ wget www.google.com
```

7. Stop and destroy the Tracing session

```
$ sudo lttng destroy
```


Installing TraceCompass

1. Installing Java

   java -version
   sudo apt update
   sudo apt install default-jre

2. Installing TraceCompass

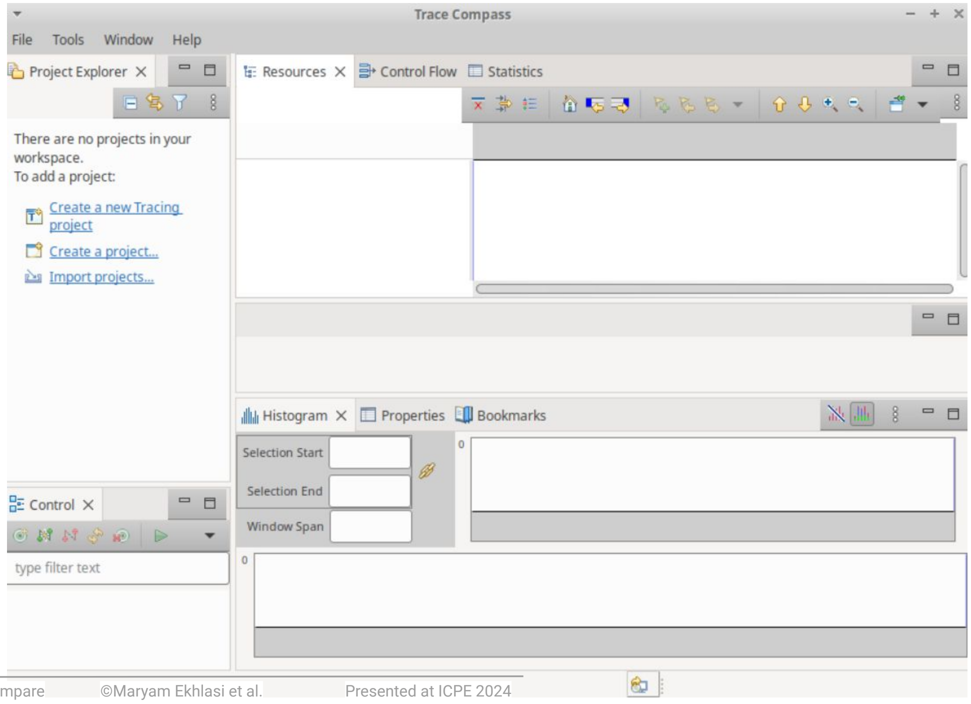
   Go to https://eclipse.dev/tracecompass/

   Tar xf <name>

   Cd trace-compass

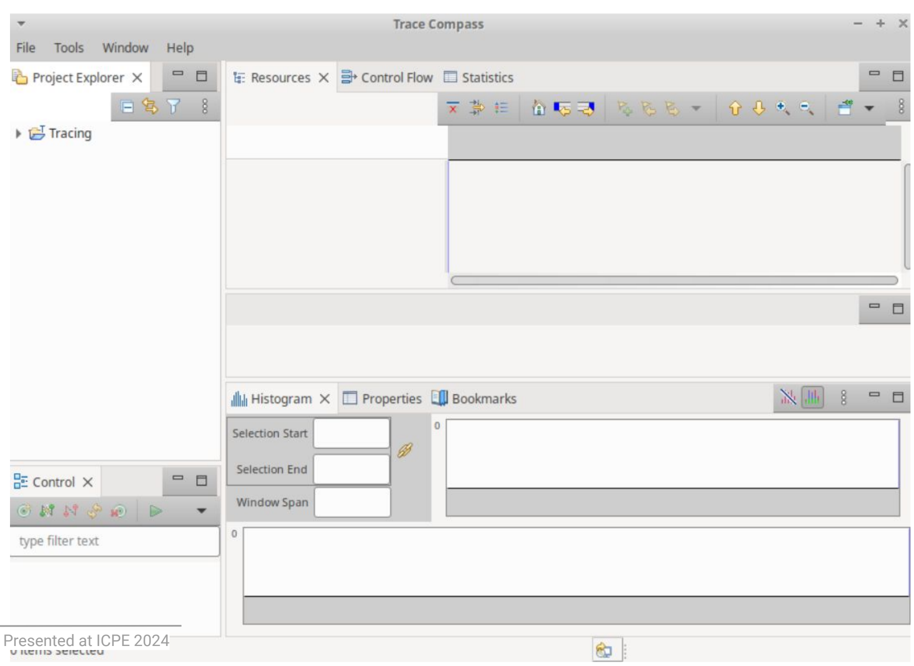
   ./tracecompass

Empty space TraceCompass

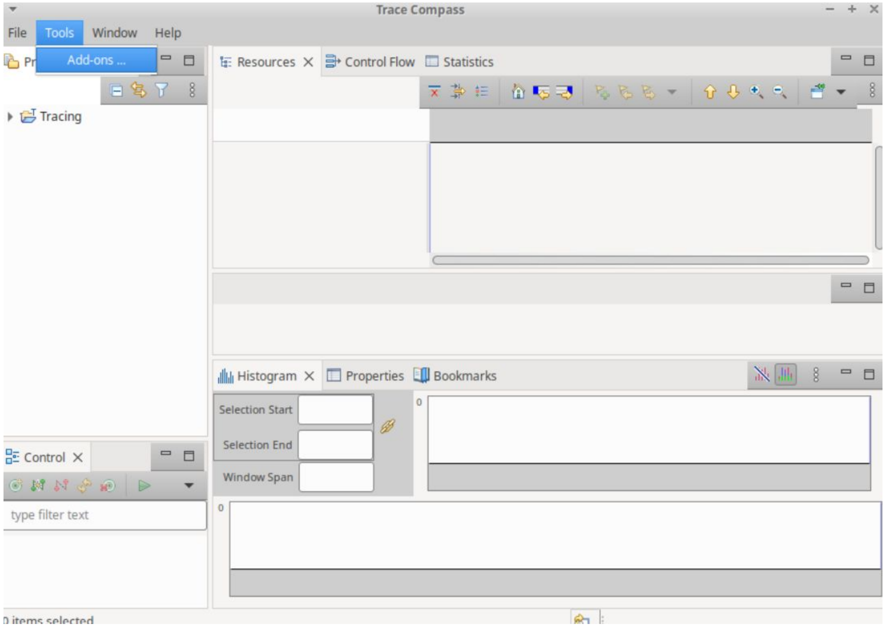


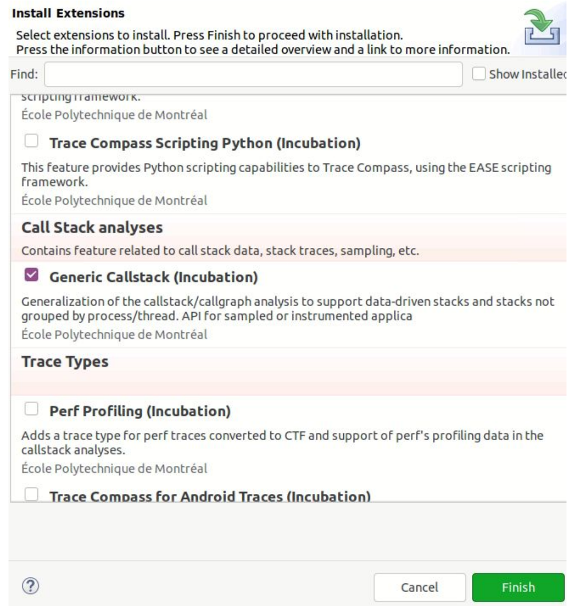
Create a tracing project

File >> new tracing project

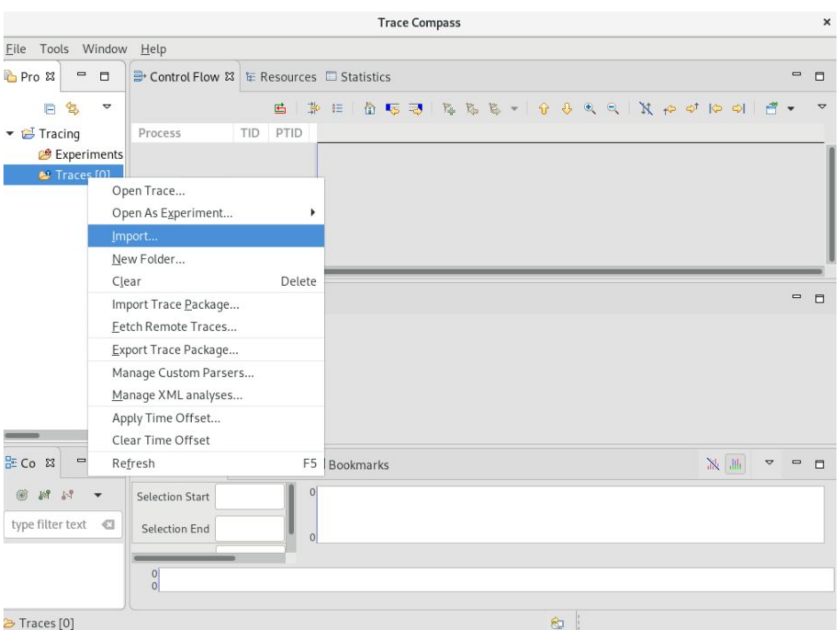


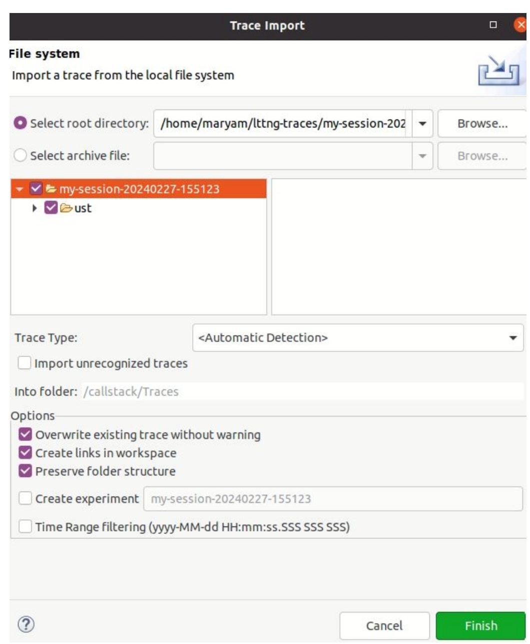
Install the required add-ons



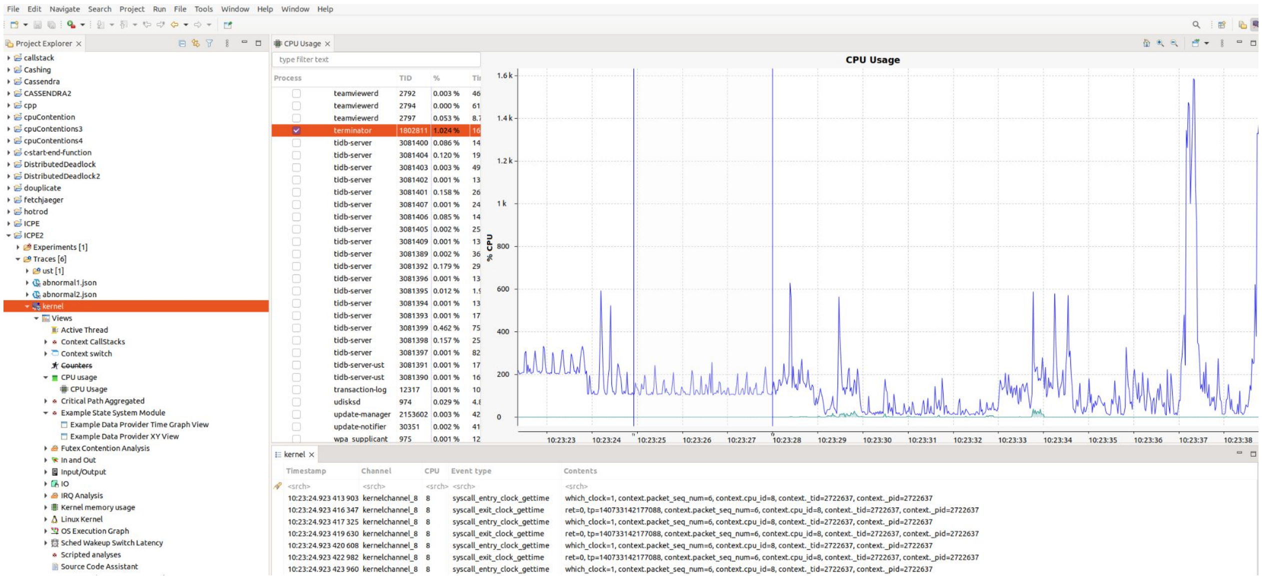


Import traces





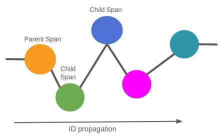
View results

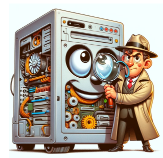


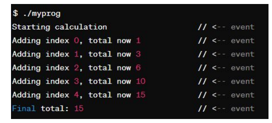
Source codes are available for free!

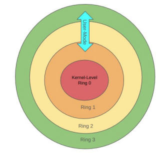
1. https://github.com/eclipse-tracecompass-incubator/org.eclipse.tracecompass.incubator
2. https://git.eclipse.org/r/c/tracecompass.incubator/org.eclipse.tracecompass.incubator/+/196496

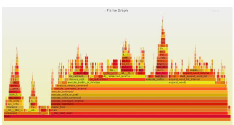
Thank you!

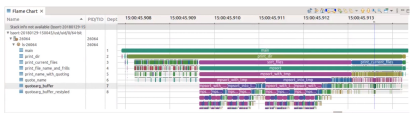


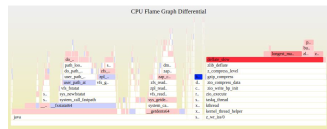


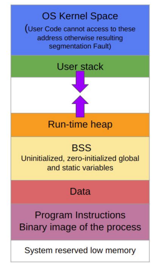


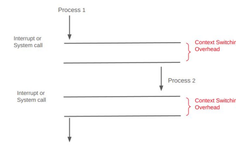












References

1. https://lttng.org/docs/v2.13/
2. https://eclipse.dev/tracecompass/
3. https://github.com/dorsal-lab/Tracevizlab/
4. https://www.brendangregg.com/
5. Maryam Ekhlasi, Fatemeh Faraji Daneshgar, Michel Dagenais, et al. DTraComp: Comparing distributed execution traces for understanding intermittent latency sources. Authorea. October 18, 2023.