Linux Observability and Tuning using bpftrace

Module 5: Advanced Scripting & Production Use

Instructor: Chandrashekar Babu <training@chandrashekar.info>

Module Overview: What You'll Learn

Advanced Data Structure Access

Techniques for safely traversing complex kernel structures and extracting meaningful data for observability

Production-Ready Scripts

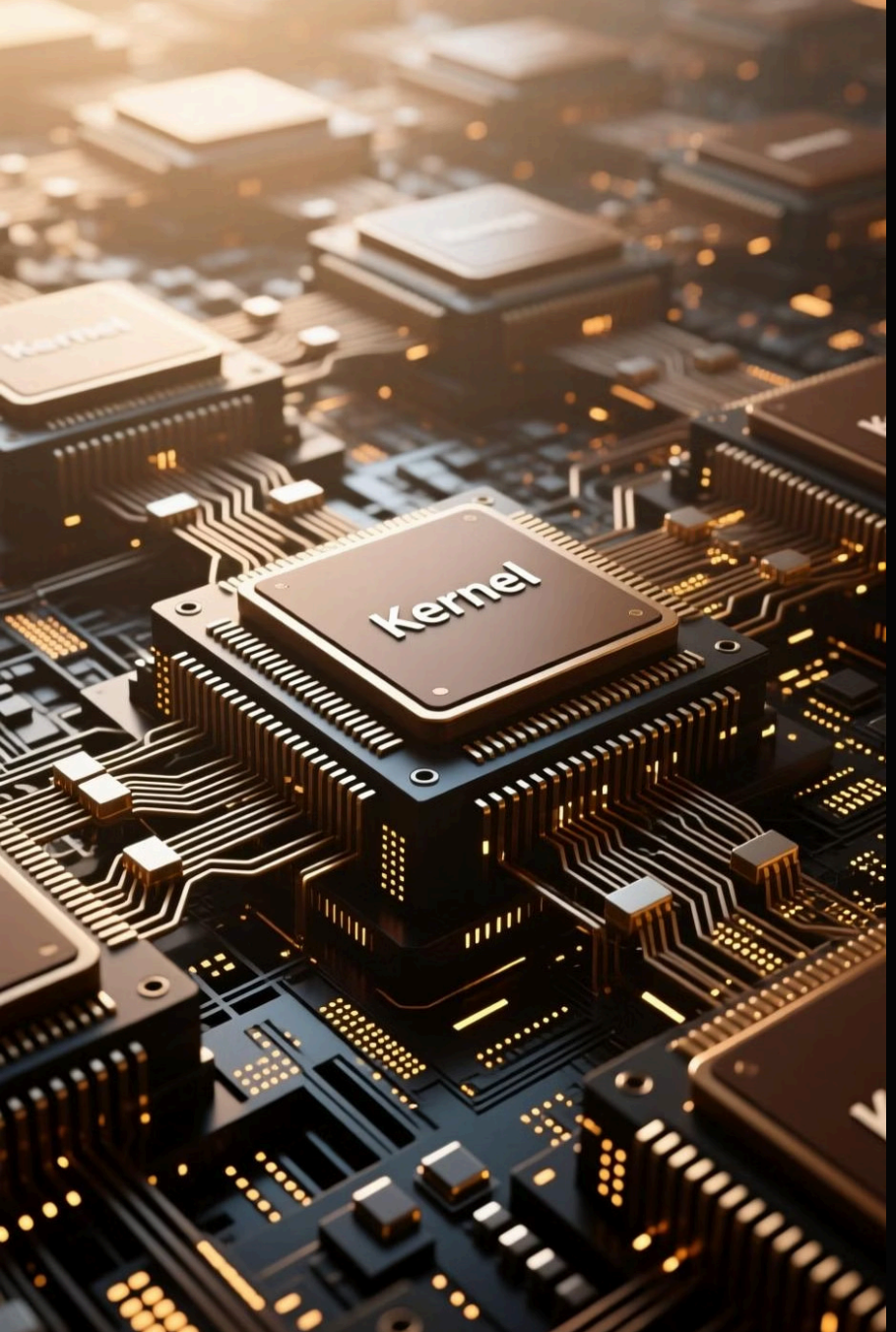
Methods to create reliable, performant bpftrace scripts that can run continuously in production environments

Container-Aware Observability

Strategies for isolating and targeting specific containerised workloads with precise tracing

Real-World Troubleshooting

Practical application of bpftrace for solving complex production issues in high-pressure scenarios



Part 1: Advanced Kernel Data Structures

Kernel Structure Access Challenges

Working with kernel structures presents unique challenges:

- **ABI Stability:** Kernel structures change between versions; field offsets shift
- **Nested Pointers:** Following multiple pointer references requires careful null checking
- **Type Information:** BTF (BPF Type Format) required for complex structs
- **Memory Safety:** Accessing invalid memory can crash the kernel

bpftrace provides safety mechanisms, but requires careful programming to maintain stability across kernel versions.

> ⚠ **WARNING:** Indiscriminate kernel memory access in production can lead to system instability and even crashes. Always validate scripts in a test environment first.

For maximum compatibility, use BTF where available, and provide fallback mechanisms for older kernels.

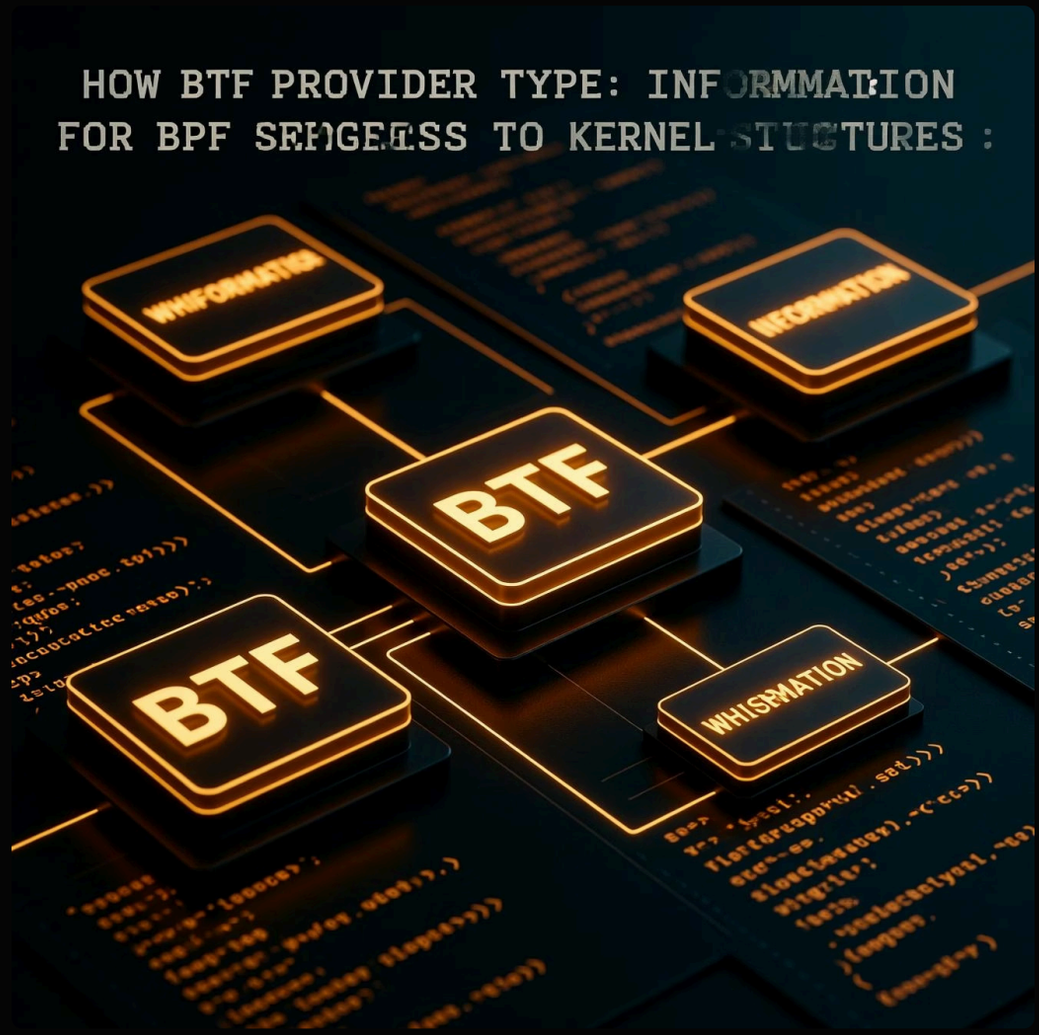
BTF: BPF Type Format

BTF provides critical type information for kernel structures:

- Available since kernel 5.4 (full support in 5.10+)
- Enables direct field access using `->` notation
- Simplifies complex struct traversal with type checking
- Retains field names across kernel updates

Check BTF availability on your system:

```
$ ls -la /sys/kernel/btf/vmlinux
$ bpftrace -e 'BEGIN { exit() }' 2>&1 | grep BTF
```



BTF provides a metadata layer that allows bpftrace to safely access kernel structures without hardcoded offsets.

Struct Traversal: Direct vs. Manual

Direct Access (BTF Available)

```
// Modern approach with BTF
bpftrace -e '
tracepoint:sched:sched_process_exec {
  $task = (struct task_struct *)curtask;
  $mm = $task->mm;
  if ($mm != 0) {
    printf("Process: %s, VMSize: %lu KB\n",
      comm, $mm->total_vm * 4);
  }
}'
```

Manual Access (No BTF)

```
// Legacy approach for older kernels
bpftrace -e '
#ifndef BPFTRACE_HAVE_BTF
#include <linux/sched.h>
#include <linux/mm_types.h>
#endif
tracepoint:sched:sched_process_exec {
 $task = (struct task_struct *)curtask;
 $mm_offset = offsetof(struct task_struct, mm);
 $mm = (struct mm_struct *)
 *((void **)((char *)$task + $mm_offset));
 if ($mm != 0) {
    $vm_offset = offsetof(struct mm_struct,
                          total_vm);
    $total_vm = *((uint64_t *)
    ((char *)$mm + $vm_offset));
    printf("Process: %s, VMSize: %lu KB\n",
                    comm, $total_vm * 4);
 }
}'
```

Key Kernel Structures for Observability

task_struct

Central process structure containing execution state, memory info, scheduling data, and relationship to other processes

Access via: `curtask` or `task` variable in many contexts

mm_struct

Process memory descriptor with virtual memory mappings, heap/stack info, and memory statistics

Access via: `task->mm` (NULL for kernel threads)

socket

Network socket information, protocol data, connection state, and buffer statistics

Access through `sock` in networking kprobes

file / inode / dentry

Filesystem structures with metadata, permissions, and relationships

Access in VFS kprobes and tracepoints

Hands-On: task_struct Traversal

Let's explore process ancestry using struct traversal:

```
#!/usr/bin/env bpftrace

BEGIN {
  printf("%-16s %-7s %-7s %-7s %s\n",
    "COMM", "PID", "PPID", "GPID", "GGPID");
}

tracepoint:sched:sched_process_fork {
  $task = (struct task_struct *)curtask;
  $parent = $task->parent;
  $grandparent = $parent->parent;
  $great_grandparent = $grandparent->parent;

  printf("%-16s %-7d %-7d %-7d %d\n",
    $task->comm,
    $task->tgid,
    $parent->tgid,
    $grandparent->tgid,
    $great_grandparent->tgid);
}
```

Exercise: Modify this script to:

1. Handle cases where a process has fewer than 3 generations of parents
2. Display the start time of each process in the hierarchy
3. Filter to show only processes with deep ancestry (4+ levels)

Save as `task_ancestry.bt` and run with `sudo bpftrace task_ancestry.bt`.

Hint: Use null pointer checks before dereferencing each parent.

File Descriptor & VFS Traversal

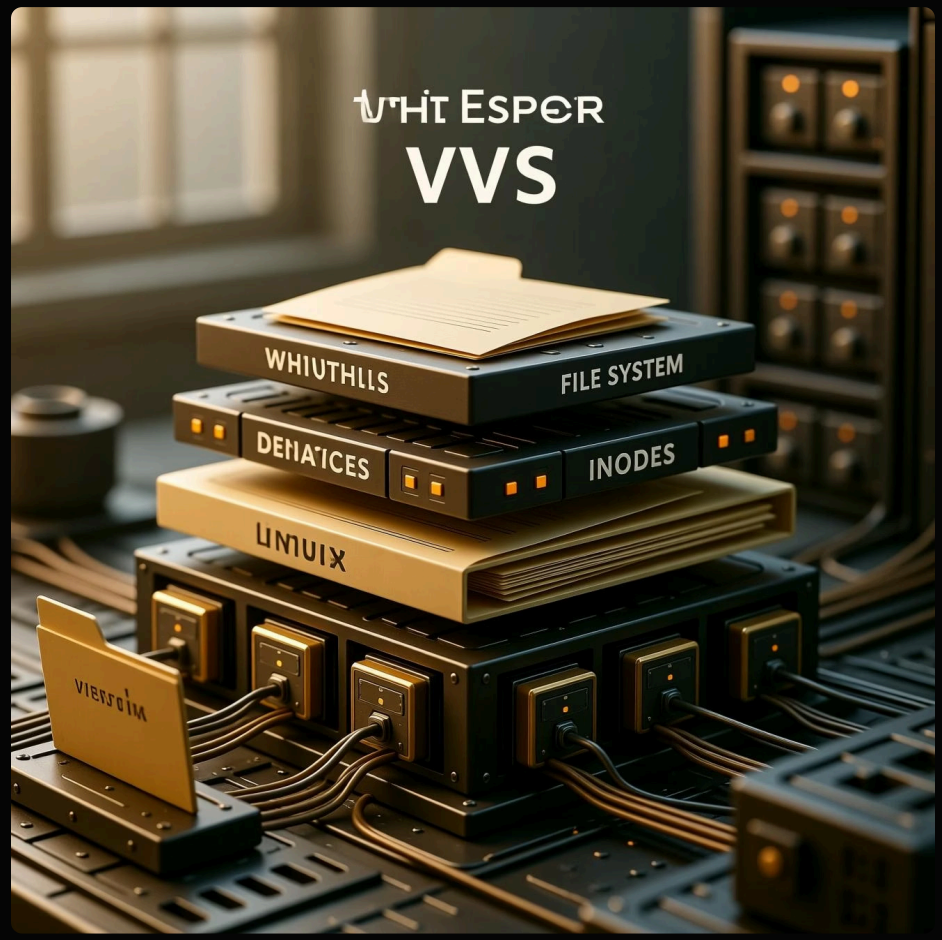
Tracing file operations requires navigating complex VFS structures:

```
// Trace file opens with full path resolution
bpftrace -e '
kprobe:do_filp_open {
  $pwd = (struct path *)arg0;
  $dentry = $pwd->dentry;
  $parent = $dentry->d_parent;

  // Get filename from dentry
  $filename = str($dentry->d_name.name);

  // Get parent directory name
  $dirname = str($parent->d_name.name);

  printf("Process %s opening %s/%s\n",
    comm, $dirname, $filename);
}'
```



The VFS (Virtual File System) layer abstracts underlying filesystems through a series of nested structures. Understanding these relationships is crucial for effective I/O tracing.

Custom Struct Definitions

For kernel structs not directly accessible or when working with application-specific types:

```
// Define a custom struct for socket monitoring
#!/usr/bin/env bpftrace

struct sock_data_t {
  u32 pid;
  u32 saddr;
  u32 daddr;
  u16 sport;
  u16 dport;
  u32 state;
  u64 tx_bytes;
  u64 rx_bytes;
};

// Map to store socket information
BPF_HASH(sock_stats, u64, struct sock_data_t);

// Track TCP socket creation
kprobe:tcp_connect {
  $sk = (struct sock *)arg0;
  $inet_sk = (struct inet_sock *)$sk;

  $key = (u64)$sk;
  sock_stats[$key].pid = pid;
  sock_stats[$key].saddr = $inet_sk->inet_saddr;
  sock_stats[$key].daddr = $inet_sk->inet_daddr;
  sock_stats[$key].sport = $inet_sk->inet_sport;
  sock_stats[$key].dport = $inet_sk->inet_dport;
  sock_stats[$key].state = $sk->__sk_common.skc_state;
}
```

This approach creates a shadow copy of the kernel data, allowing for efficient tracking of socket state changes.

Field Offset Determination

Using BTF Information

```
bpftrace -e '
BEGIN {
  printf("task_struct.mm offset: %d\n",
    offsetof(struct task_struct, mm));
  printf("mm_struct.total_vm offset: %d\n",
    offsetof(struct mm_struct, total_vm));
  exit();
}'
```

This dynamic approach ensures compatibility across kernel versions by querying BTF at runtime.

Manual Discovery via pahole

For systems without BTF, use pahole tool:

```
$ pahole -C task_struct /sys/kernel/btf/vmlinux | less

struct task_struct {
  /* ... */
  struct mm_struct *       mm;             /*    752 8 */
  /* ... */
};
```

Then use the discovered offset in your scripts:

```
#define TASK_MM_OFFSET 752
```

Advanced Map Techniques

Maps provide persistent storage for complex data structures:

- **BPF_HASH:** Key-value store for arbitrary data
- **BPF_ARRAY:** Fixed-size array for fast lookup
- **BPF_HISTOGRAM:** Built-in distribution visualisation
- **BPF_PERF_OUTPUT:** Ring buffer for data streaming

Using maps with custom structs enables sophisticated analysis patterns not possible with simpler tools.

```
// Track per-process memory changes
struct mem_info_t {
 u64 last_vm_size;
 u64 growth_count;
 u64 total_growth;
};

BPF_HASH(process_mem, u32, struct mem_info_t);

tracepoint:sched:sched_process_exit {
 $pid = args->pid;

 if (process_mem.lookup(&$pid)) {
 $info = process_mem.lookup(&$pid);
 printf("PID %d exited: %d growth events, avg: %d KB\n",
 $pid, $info->growth_count,
 $info->growth_count > 0 ?
 $info->total_growth / $info->growth_count : 0);
 process_mem.delete(&$pid);
 }
}
```

Part 2: Production-Grade Scripting

What Makes a Script Production-Ready?

Efficiency

Minimal overhead, limited event collection rate, sampling where appropriate

Safety

Memory access guards, error handling, graceful failures

Readability

Clear structure, comments, consistent naming conventions

Reliability

Handles edge cases, version differences, unexpected inputs

Configurability

Parameterised with environment variables or command-line arguments

Long-Running Scripts: Ring Buffers

Standard bpftrace output is unsuitable for continuous operation. Use perf buffers instead:

```
#!/usr/bin/env bpftrace

struct event_t {
  u64 ts;
  u32 pid;
  char comm[16];
  u64 delta_ms;
};

// Create perf output buffer
BPF_PERF_OUTPUT(events);

// Track when processes enter syscalls
kprobe:sys_enter_read {
  // Store entry timestamp in map
  @start[tid] = nsecs;
}

// Measure duration when syscalls complete
kretprobe:sys_exit_read / @start[tid] / {
  $delta = nsecs - @start[tid];
  $delta_ms = $delta / 1000000;

  // Only report slow operations (>10ms)
  if ($delta_ms > 10) {
    // Populate event structure
    struct event_t event = {
      .ts = nsecs,
      .pid = pid,
      .delta_ms = $delta_ms
    };
    bpf_probe_read_str(&event.comm, sizeof(event.comm),
                       comm);

    // Send to ring buffer
    events.perf_submit(args, &event, sizeof(event));
  }

  // Clean up entry map
  delete(@start[tid]);
}
```

Consuming Ring Buffer Data

Outside bpftrace, you need a consumer process to read from the ring buffer:

```
#!/usr/bin/python3
from bcc import BPF
import json
import datetime
import signal
import sys

# Load the BPF program
b = BPF(text=bpf_text)

# Process events from ring buffer
def handle_event(cpu, data, size):
 event = b["events"].event(data)
 print(json.dumps({
 "timestamp": datetime.datetime.now().
 isoformat(),
 "pid": event.pid,
 "comm": event.comm.decode(),
 "duration_ms": event.delta_ms
 }))

# Attach event handler
b["events"].open_perf_buffer(handle_event)

# Main loop
while True:
 try:
 b.perf_buffer_poll()
 except KeyboardInterrupt:
 break
```

Error Handling Strategies

Robust scripts need comprehensive error handling to avoid silent failures:

```
// Check for null pointers before dereferencing
kprobe:tcp_connect {
  $sk = (struct sock *)arg0;
  if ($sk == 0) {
    @null_sock_count++;
    return;
  }

  $inet_sk = (struct inet_sock *)$sk;
  $saddr = $inet_sk->inet_saddr;
  // ...
}

// Use ternary operators for safe access
tracepoint:syscalls:sys_enter_execve {
  $filename = args->filename;
  printf("Executing: %s\n",
         $filename ? str($filename) : "[unknown]");
}

// Track error rates and report them
interval:s:60 {
  printf("Error counts in the last minute:\n");
  printf("  Null pointers: %d\n", @null_sock_count);
  printf("  Failed lookups: %d\n", @map_lookup_failures);
  clear(@null_sock_count);
  clear(@map_lookup_failures);
}
```

Always track error conditions in production scripts to detect unexpected behaviour patterns.

Performance Optimisation Techniques

Event Filtering

Apply filters as early as possible in the script to reduce processing overhead:

```
// Only trace processes in specific cgroup
tracepoint:syscalls:sys_enter_read
/ cgroup == cgroupid("/system.slice/app.service") /
{
  // Processing here
}
```

Map Management

Prevent unbounded growth in maps with explicit cleanup:

```
// Age out old entries periodically
interval:s:30 {
  $now = nsecs;
  $timeout_ns = 30 * 1000000000; // 30s

  // Iterate through map, delete old entries
  @start = filter(@start, {
    $delta = $now - $value;
    return $delta < $timeout_ns;
  });
}
```

Sampling

Use statistical sampling to reduce overhead for high-frequency events:

```
// Only trace 1% of events
kprobe:__alloc_skb / (rand() % 100) == 0 / {
 // 1/100 sampling rate
 @sk_buffer_sizes = hist(arg1);
}
```

Monitoring Script Resource Usage

Production scripts must be monitored for their own resource consumption:

```
// Self-monitoring bpftrace script
BEGIN {
  printf("Starting monitoring at %d\n", nsecs);
  @start_time = nsecs;

  // Record initial state
  @start_memory = kstack;
}

// Periodically report script overhead
interval:s:30 {
  $runtime_s = (nsecs - @start_time) / 1000000000;
  printf("Runtime: %d seconds\n", $runtime_s);
  printf("Maps in use: %d\n", map_elements());

  // Report map sizes
  printf("Map sizes:\n");
  printf("  @start: %d entries\n", kmap_elements(@start));
  printf("  @sock_stats: %d entries\n",
        kmap_elements(@sock_stats));

  // Print largest maps
  print(@sock_stats, 5);
}
```

External Monitoring Tools

Use Linux performance tools to monitor bpftrace impact:

```
# Track bpftrace CPU usage
$ pidstat -p $(pgrep bpftrace) 5

# Monitor memory consumption
$ ps -o pid,vsz,rss,pmem,comm -p $(pgrep bpftrace)

# Check for leaked maps or programs
$ bpftool map list
$ bpftool prog list | grep bpf
```

For critical production use, set resource limits and alarms:

```
# Run with CPU limit and timeout
$ timeout 12h cpulimit -l 5 -- \
 bpftrace -f json ./production_script.bt \
 > /var/log/bpftrace.log
```

Practical Example: File I/O Latency Monitoring

```
#!/usr/bin/env bpftrace

// Configuration via env vars (with defaults)
BEGIN {
 printf("Starting I/O latency monitoring\n");

 // Parse configuration
 $target_pid_str = str(env("TARGET_PID"));
 $target_pid = atoi($target_pid_str);
 printf("Target PID filter: %s\n",
 $target_pid > 0 ? $target_pid_str : "all");

 $threshold_ms_str = str(env("THRESHOLD_MS"));
 $threshold_ms = atoi($threshold_ms_str);
 if ($threshold_ms == 0) { $threshold_ms = 10; }
 printf("Latency threshold: %d ms\n", $threshold_ms);

 // Init start timestamp for runtime tracking
 @start_time = nsecs;
}

// Track file read/write entry
kprobe:vfs_read,
kprobe:vfs_write
/ $target_pid == 0 || pid == $target_pid /
{
 // Store entry timestamp and operation type
 @start[tid] = nsecs;
 @is_write[tid] = func == "vfs_write";

 // Capture file info
 $file = (struct file *)arg0;
 $dentry = $file->f_path.dentry;
 $filename = str($dentry->d_name.name);
 @filename[tid] = $filename;
}

// Measure duration on exit
kretprobe:vfs_read,
kretprobe:vfs_write
/ @start[tid] /
{
 $delta = nsecs - @start[tid];
 $delta_ms = $delta / 1000000;

 // Only report operations over threshold
 if ($delta_ms >= $threshold_ms) {
 $op = @is_write[tid] ? "WRITE" : "READ";
 $filename = @filename[tid];
 $bytes = retval > 0 ? retval : 0;

 printf("%s %-16s %-20s %6d ms %8d bytes\n",
 $op, comm, $filename, $delta_ms, $bytes);

 // Collect statistics for summary
 @total_slow++;
 @total_bytes += $bytes;
 @latency_sum += $delta_ms;
 @latency_by_file[$filename] += $delta_ms;
 }

 // Clean up tracking maps
 delete(@start[tid]);
 delete(@is_write[tid]);
 delete(@filename[tid]);
}

// Produce summary at exit
END {
 $runtime_s = (nsecs - @start_time) / 1000000000;
 printf("\nSummary after %d seconds:\n", $runtime_s);
 printf("Slow operations: %d\n", @total_slow);

 if (@total_slow > 0) {
 printf("Avg latency: %d ms\n", @latency_sum / @total_slow);
 printf("Total bytes: %d\n", @total_bytes);

 printf("\nLatency by file (top 10):\n");
 print(@latency_by_file, 10);
 }
}
```

Part 3: Container-Aware Tracing

Container Concepts: Namespaces & cgroups

Linux Namespaces

- **PID:** Process isolation (PIDs in container ≠ host PIDs)
- **Network:** Isolated network stack
- **Mount:** Isolated filesystem view
- **UTS:** Hostname isolation
- **IPC:** Isolated IPC resources
- **User:** User/group ID mapping
- **Cgroup:** Cgroup resource controller isolation (newer)

Containers use namespaces to create isolation boundaries, but these boundaries create challenges for tracing.

Control Groups (cgroups)

- Resource limiting (CPU, memory, I/O, etc.)
- Prioritisation
- Accounting
- Control

cgroups provide a consistent way to identify containers regardless of namespace complexity. Docker and Kubernetes use cgroups to manage resource constraints.

bpftrace can filter by cgroup ID to target specific containers without being inside their namespaces.

Identifying Container IDs and cgroups

To trace containers, first map container IDs to cgroup IDs:

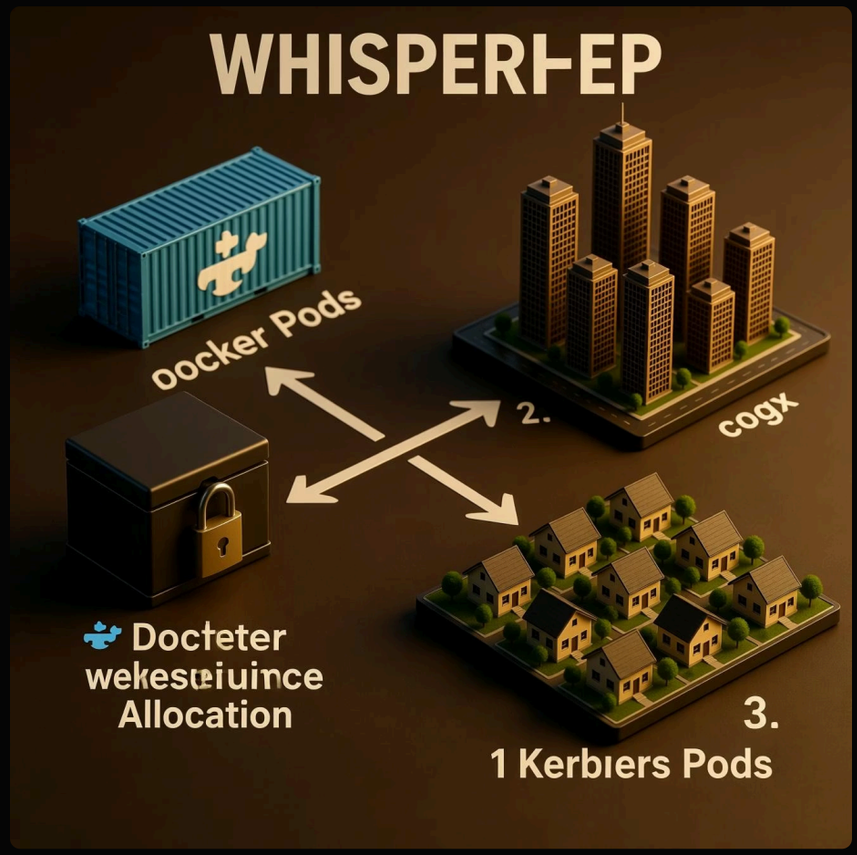
```
# Docker containers
$ docker inspect --format \
  '{{.Id}} {{.HostConfig.CgroupParent}}' \
  $(docker ps -q)

# Kubernetes pods
$ for pod in $(kubectl get pods -o name); do
    echo "Pod: $pod"
    kubectl get "$pod" -o json | \
      jq '.metadata.uid,
          .status.containerStatuses[].containerID'
  done

# Find cgroup paths
$ systemd-cgls
```

Get the cgroup ID for use in bpftrace:

```
$ cat /proc/self/cgroup
$ bpftrace -e 'BEGIN {
  printf("Cgroup ID: %u\n",
    cgroupid("/sys/fs/cgroup/system.slice/docker-[ID].scope"));
  exit();
}'
```



Container runtimes use cgroups to manage resources, creating a hierarchical structure of control groups. Each Docker container gets its own cgroup, while Kubernetes pods may group multiple containers in shared cgroups.

Filtering by Container (cgroup)

```
#!/usr/bin/env bpftrace

// Usage: sudo ./container_io.bt [container_cgroup_id]

BEGIN {
 printf("Container I/O Monitoring\n");

 // Parse cgroup ID from args
 $target_cgroupid_str = str($1);
 if ($target_cgroupid_str != "") {
 $target_cgroupid = strtoul($target_cgroupid_str, 16);
 printf("Filtering for cgroup ID: %u\n", $target_cgroupid);
 } else {
 $target_cgroupid = 0;
 printf("Monitoring all cgroups\n");
 }
}

// Track file read/write operations
kprobe:vfs_read,
kprobe:vfs_write
/ $target_cgroupid == 0 || cgroup == $target_cgroupid /
{
 $is_write = (func == "vfs_write");
 @io_count[$is_write ? "write" : "read"]++;

 // Get file info
 $file = (struct file *)arg0;
 $path = $file->f_path.dentry->d_name.name;

 // Update bytes counters by file
 @bytes_by_file[$is_write ? "write" : "read",
 comm, str($path)] += arg2;

 // Update container identification
 @container_ids[cgroup, comm]++;
}

// Print summary every 10 seconds
interval:s:10 {
 time("%H:%M:%S ");
 printf("I/O Operations: reads=%d, writes=%d\n",
 @io_count["read"],
 @io_count["write"]);

 printf("\nTop files by bytes:\n");
 print(@bytes_by_file, 10);

 clear(@io_count);
 clear(@bytes_by_file);
}
```

Container Network Visibility

Network namespaces make container traffic monitoring challenging. Use bpftrace with kernel network hooks:

```
#!/usr/bin/env bpftrace

BEGIN {
  printf("Container Network Monitoring\n");

  // Parse container cgroup ID from arg
  $target_cgroupid = strtoul(str($1), 16);
  printf("Target cgroup: %u\n", $target_cgroupid);
}

// Track TCP connections
kprobe:tcp_connect
/ cgroup == $target_cgroupid /
{
  $sk = (struct sock *)arg0;
  $inet_fam = $sk->__sk_common.skc_family;

  // Only IPv4 for simplicity
  if ($inet_fam == AF_INET) {
    $daddr = ntop($sk->__sk_common.skc_daddr);
    $dport = $sk->__sk_common.skc_dport;

    // Convert port from network to host order
    $dport = (($dport & 0xFF) << 8) | ($dport >> 8);

    printf("TCP connect: %s:%d -> %s:%d\n",
      comm, pid, $daddr, $dport);

    // Track connection start time
    @conn_start[pid, $daddr, $dport] = nsecs;
  }
}
```

This can be extended to track connection duration, bytes transferred, and more:

```
// Track successful connections
kretprobe:tcp_connect
/ @conn_start[pid, "*", 0] /
{
 $ret = retval;
 if ($ret == 0) {
 // Connection successful
 @conn_status[pid, "success"]++;
 } else {
 // Connection failed
 @conn_status[pid, "failed"]++;
 @conn_errors[pid, - $ret]++;
 }
}

// Track data transfer
kprobe:tcp_sendmsg
/ cgroup == $target_cgroupid /
{
 $sk = (struct sock *)arg0;
 $inet_fam = $sk->__sk_common.skc_family;
 if ($inet_fam == AF_INET) {
 $daddr = ntop($sk->__sk_common.skc_daddr);
 $dport = $sk->__sk_common.skc_dport;
 $dport = (($dport & 0xFF) << 8) | ($dport >> 8);
 $size = arg2;

 @bytes_sent[$daddr, $dport] += $size;
 }
}
```

Cross-Container Visibility

Monitoring communication between containers requires correlating events across cgroups:

```
#!/usr/bin/env bpftrace

#include
#include

// Store container name/ID mapping
BEGIN {
  @container_map[cgroupid("/sys/fs/cgroup/system.slice/app-frontend.service")] =
"frontend";
  @container_map[cgroupid("/sys/fs/cgroup/system.slice/app-backend.service")] =
"backend";
  @container_map[cgroupid("/sys/fs/cgroup/system.slice/app-database.service")]
= "database";
}

// Track socket operations
kprobe:sock_sendmsg,
kprobe:sock_recvmsg
{
  $sock = (struct socket *)arg0;
  $sk = $sock->sk;

  if ($sk) {
    // Get container name from cgroup id
    $cgroup = cgroup;
    $container = @container_map[$cgroup];
    if ($container == "") {
      $container = "unknown";
    }

    // Record operation by container
    $op = func == "sock_sendmsg" ? "send" : "recv";
    @socket_ops[$container, $op]++;

    // Get peer info if available
    if ($sk->__sk_common.skc_state == TCP_ESTABLISHED) {
      $daddr = ntop($sk->__sk_common.skc_daddr);
      $dport = $sk->__sk_common.skc_dport;
      $dport = (($dport & 0xFF) << 8) | ($dport >> 8);

      // Track communication between containers
      @conn_counts[$container, $daddr, $dport]++;
    }
  }
}
```

Cross-container visibility is essential for microservice architecture troubleshooting, allowing you to trace request flows across service boundaries.

Namespace Awareness: PID Translation

PIDs in containers differ from host PIDs due to PID namespaces. This script demonstrates PID translation:

```
#!/usr/bin/env bpftrace

#include
#include

BEGIN {
  printf("PID Namespace Translation\n");
  printf("%-6s %-6s %-16s %s\n",
        "HPID", "CPID", "COMM", "CONTAINER");
}

// Triggered on exec to capture process starts
tracepoint:sched:sched_process_exec {
  $task = (struct task_struct *)curtask;

  // Get host PID (what we see outside container)
  $host_pid = $task->pid;

  // Get the task's PID namespace
  $nsproxy = $task->nsproxy;
  if ($nsproxy) {
    $pid_ns = $nsproxy->pid_ns_for_children;

    // Get container PID (what process sees inside container)
    $container_pid = $task->thread_pid->numbers[0].nr;

    // Try to identify container from cgroup
    $cgroup_id = cgroup;
    if (@container_map[$cgroup_id] != "") {
      $container = @container_map[$cgroup_id];
    } else {
      $container = "unknown";
    }

    printf("%-6d %-6d %-16s %s\n",
          $host_pid, $container_pid, comm, $container);

    // Store mapping for future reference
    @pid_map[$host_pid] = $container_pid;
    @container_map[$cgroup_id] = $container;
  }
}
```

This capability is critical for mapping container-reported issues back to host-visible processes for investigation.

Container Resource Usage Monitoring

CPU Accounting

```
// Per-container CPU usage tracking
profile:hz:99
{
  $cgroup_id = cgroup;
  @cpu[$cgroup_id]++;

  // Track per-thread CPU within container
  if ($target_cgroupid == $cgroup_id) {
    @cpu_by_thread[tid, comm]++;
  }
}

interval:s:5 {
  printf("CPU Usage by Container:\n");

  // Calculate percentages
  $total = 0;
  map_foreach(@cpu, $k, $v) {
    $total += $v;
  }

  // Print normalized to 100%
  map_foreach(@cpu, $cg, $count) {
    $container = @container_map[$cg] != "" ?
                 @container_map[$cg] :
                 "unknown";
    $pct = ($count * 100) / $total;
    printf("  %-20s: %5.2f%%\n",
           $container, $pct);
  }

  clear(@cpu);
}
```

Memory Tracking

```
// Track memory allocations by container
kprobe:kmem_cache_alloc,
kprobe:kmalloc
{
 $size = arg1;
 $cgroup_id = cgroup;

 // Track container memory
 @mem_alloc[$cgroup_id] += $size;

 // Detailed tracking for target container
 if ($target_cgroupid == $cgroup_id) {
 @mem_by_stack[$cgroup_id, kstack] += $size;
 @mem_by_thread[$cgroup_id, tid, comm] += $size;
 }
}

kprobe:kfree,
kprobe:kmem_cache_free
{
 // Memory frees are more complex to attribute
 // This is simplified tracking
 $cgroup_id = cgroup;
 @mem_free[$cgroup_id]++;
}

interval:s:10 {
 printf("Memory by Container (bytes):\n");
 print(@mem_alloc);

 if ($target_cgroupid != 0) {
 printf("\nTop Allocation Stacks:\n");
 print(@mem_by_stack, 5);
 }

 clear(@mem_alloc);
 clear(@mem_free);
 // Don't clear stack map to track leaks
}
```

Hands-On: Container-Aware Disk I/O Latency Tool

```
#!/usr/bin/env bpftrace

// Container-aware I/O latency monitoring tool
// Usage: sudo ./container_io_latency.bt [container_name] [threshold_ms]

BEGIN {
 printf("Container I/O Latency Monitoring\n");

 // Parse container name from args
 $container_name = str($1);
 if ($container_name == "") {
 $container_name = "all";
 }
 printf("Target container: %s\n", $container_name);

 // Parse latency threshold
 $threshold_ms = atoi(str($2));
 if ($threshold_ms == 0) {
 $threshold_ms = 10; // Default: 10ms
 }
 printf("Latency threshold: %d ms\n", $threshold_ms);

 // Look up cgroup ID for container name
 if ($container_name != "all") {
 $cmd = "find /sys/fs/cgroup -name \"*" + $container_name + "*\" | head -1";
 $cgpath = system($cmd);
 if ($cgpath != "") {
 $target_cgroupid = cgroupid($cgpath);
 printf("Cgroup ID: %u, Path: %s\n",
 $target_cgroupid, $cgpath);
 } else {
 printf("Container not found, monitoring all I/O\n");
 $target_cgroupid = 0;
 }
 } else {
 $target_cgroupid = 0;
 }
}

// Track block I/O request start time
kprobe:blk_mq_start_request
{
 $req = (struct request *)arg0;
 $start_time[arg0] = nsecs;
}

// Track completion and latency
kprobe:blk_account_io_done
/ $start_time[arg0] /
{
 $req = (struct request *)arg0;
 $delta = nsecs - $start_time[arg0];
 $delta_ms = $delta / 1000000;

 // Skip if under threshold
 if ($delta_ms < $threshold_ms) {
 delete($start_time[arg0]);
 return;
 }

 // Skip if container ID doesn't match (when filtering)
 $task_cgroupid = cgroup;
 if ($target_cgroupid != 0 && $task_cgroupid != $target_cgroupid) {
 delete($start_time[arg0]);
 return;
 }

 // Get operation type and size
 $op = $req->cmd_flags & 1 ? "write" : "read";
 $bytes = $req->__data_len;

 // Report slow I/O
 time("%H:%M:%S ");
 printf("%s I/O: %s %d bytes, latency: %d ms\n",
 $container_name != "all" ? $container_name : comm,
 $op, $bytes, $delta_ms);

 // Collect statistics
 @io_count[$task_cgroupid, $op]++;
 @io_bytes[$task_cgroupid, $op] += $bytes;
 @io_latency[$task_cgroupid, $op] += $delta_ms;

 // Histogram for analysis
 @latency_hist[$op] = hist($delta_ms);

 delete($start_time[arg0]);
}

// Print summary periodically
interval:s:30 {
 printf("\n=== I/O Latency Summary ===\n");

 // Aggregate by container
 printf("By Container/Operation:\n");
 printf("%-20s %-5s %10s %10s %10s\n",
 "CONTAINER", "OP", "COUNT", "BYTES", "AVG_LAT");

 map_foreach(@io_count, $key, $count) {
 $cg = $key[0];
 $op = $key[1];
 $bytes = @io_bytes[$key[0], $key[1]];
 $latency = @io_latency[$key[0], $key[1]];
 $avg_lat = $count > 0 ? $latency / $count : 0;

 // Look up container name from cgroup ID
 $cmd = "basename $(find /sys/fs/cgroup -name '*' -type d | grep " + $cg + ")";
 $container = system($cmd);
 if ($container == "") {
 $container = "unknown-" + $cg;
 }

 printf("%-20s %-5s %10d %10d %10d\n",
 $container, $op, $count, $bytes, $avg_lat);
 }

 printf("\nLatency Distributions:\n");
 print(@latency_hist);

 // Reset for next interval
 clear(@latency_hist);
}
```

Part 4: Real-World Troubleshooting

Production Troubleshooting Workflow

Identify Symptoms

Define what's wrong - latency, errors, resource usage, crashes

Narrow Scope

Isolate affected subsystems, containers, or processes

Instrument & Measure

Deploy targeted bpftrace scripts to gather relevant data

Analyze Patterns

Look for correlations, outliers, and unexpected behavior

Test Hypothesis

Verify root cause with targeted experiments

The most effective troubleshooting combines traditional tools (metrics, logs) with targeted eBPF-based instrumentation. bpftrace is particularly valuable when existing monitoring doesn't provide sufficient visibility.

Case Study 1: Mysterious Network Latency

Scenario

A microservice application suddenly shows intermittent high latency in production. Standard monitoring shows normal CPU, memory, and disk I/O, but network requests between services are sometimes delayed by 200-500ms.

Investigation Approach

```
#!/usr/bin/env bpftrace

// Track TCP retransmissions by connection
kprobe:tcp_retransmit_skb {
  $sk = (struct sock *)arg0;
  $inet_fam = $sk->__sk_common.skc_family;

  if ($inet_fam == AF_INET) {
    $saddr = ntop($sk->__sk_common.skc_rcv_saddr);
    $daddr = ntop($sk->__sk_common.skc_daddr);
    $sport = $sk->__sk_common.skc_num;
    $dport = $sk->__sk_common.skc_dport;
    $dport = (($dport & 0xFF) << 8) | ($dport >> 8);

    // Identify container
    $cgroup_id = cgroup;
    $container = @container_map[$cgroup_id];
    if ($container == "") {
      $container = "unknown";
    }

    printf("TCP retransmit: %s:%d -> %s:%d (%s)\n",
           $saddr, $sport, $daddr, $dport, $container);

    @retransmits[$saddr, $daddr, $sport, $dport]++;
    @container_retransmits[$container]++;
  }
}
```

Root Cause

Analysis revealed TCP retransmits occurring due to network packet drops between specific services. Further investigation with bpftrace identified packets being dropped by iptables rules:

```
// Track packet drops from iptables
kprobe:ipt_do_table {
  @iptables_checks++;
}

kretprobe:ipt_do_table {
  if (retval == -NF_DROP) {
    printf("Packet dropped by iptables\n");
    @iptables_drops++;
    @drop_stack[kstack()] = count();
  }
}
```

The root cause was a recently deployed network policy that accidentally blocked some legitimate traffic between services. Fixing the network policy immediately resolved the issue.

Case Study 2: Memory Pressure in Containers

Scenario

A critical application container is being OOM-killed despite having adequate memory limits. The container's memory usage appears normal in metrics, yet it's terminated every few hours.

Investigation Approach

```
#!/usr/bin/env bpftrace

// Track memory reclaim pressure
kprobe:try_to_free_pages {
  @reclaim_calls++;
  @reclaim_by_cgroup[cgroup]++;

  // Track stack for high reclaim activity
  if (cgroup == $target_cgroup) {
    @reclaim_stack[kstack()] = count();
  }
}

// Track allocation failures
kretprobe:__alloc_pages_nodemask
/ retval == 0 /
{
  @alloc_failures++;
  @failure_order[arg1]++;

  if (cgroup == $target_cgroup) {
    printf("Page allocation failure: order %d\n",
           arg1);
    print(kstack());
  }
}
```

Root Cause

The investigation revealed memory fragmentation rather than overall memory pressure was the issue. The application was allocating large contiguous memory blocks, but the container's memory was fragmented:

```
// Track memory fragmentation
interval:s:10 {
  // Read /proc/buddyinfo for each NUMA node
  system("cat /proc/buddyinfo");

  // Count free pages by order in target cgroup
  $cmd = "cat /sys/fs/cgroup/memory/docker/" +
         $container_id +
         "/memory.stat | grep total_";
  system($cmd);
}
```

The root cause was a JVM with inefficient GC settings that weren't compacting memory properly. Adjusting the GC parameters to trigger more frequent compactions resolved the issue without increasing memory limits.

Case Study 3: Slow Filesystem Operations

Scenario

Application logs show intermittent high latency for file operations on a shared persistent volume. The issue occurs across multiple containers using the same storage.

Investigation Approach

```
#!/usr/bin/env bpftrace

// Track filesystem operations with context
kprobe:vfs_read,
kprobe:vfs_write,
kprobe:vfs_fsync
{
  @start[tid] = nsecs;
  @func[tid] = func;

  // Capture file info
  $file = (struct file *)arg0;
  $dentry = $file->f_path.dentry;
  @filename[tid] = str($dentry->d_name.name);

  // Get filesystem type
  $sb = $dentry->d_sb;
  if ($sb) {
    @fs_type[tid] = str($sb->s_type->name);
  }
}

// Measure completion time
kretprobe:vfs_read,
kretprobe:vfs_write,
kretprobe:vfs_fsync
/ @start[tid] /
{
  $delta = nsecs - @start[tid];
  $delta_ms = $delta / 1000000;

  // Only report slow operations
  if ($delta_ms > 50) {
    printf("%s on %s (%s) took %d ms\n",
           @func[tid], @filename[tid],
           @fs_type[tid], $delta_ms);

    // Track by filesystem type
    @latency_by_fs[@fs_type[tid], @func[tid]] =
      hist($delta_ms);
  }

  delete(@start[tid]);
  delete(@func[tid]);
  delete(@filename[tid]);
  delete(@fs_type[tid]);
}
```

Root Cause Analysis

The script revealed that fsync operations were particularly slow, especially when multiple containers performed them simultaneously. Further analysis with bpftrace showed lock contention:

```
// Track filesystem locks
kprobe:mutex_lock {
  @lock_start[tid] = nsecs;
  @lock_addr[tid] = arg0;
}

kretprobe:mutex_lock {
  $delta = nsecs - @lock_start[tid];
  $delta_ms = $delta / 1000000;

  if ($delta_ms > 10) {
    printf("Slow lock acquisition: %d ms\n",
           $delta_ms);
    @slow_locks[@lock_addr[tid]] = hist($delta_ms);
    @lock_stacks[@lock_addr[tid]] = kstack();
  }

  delete(@lock_start[tid]);
  delete(@lock_addr[tid]);
}
```

The issue was NFS locking with multiple containers accessing the same files. Switching to a more container-friendly storage solution resolved the issue.

Lab Challenge: Production Troubleshooting

You're given a multi-container application showing performance issues. Your task is to use bpftrace to identify and diagnose the root cause.

Setup

```
# Clone the lab repository
git clone https://github.com/bpftrace-workshop/production-challenge
cd production-challenge

# Start the application environment
./setup.sh

# Verify the application is running with issues
./check_status.sh
```

Challenge Tasks

1. Identify which components are experiencing performance problems
2. Use bpftrace to gather detailed performance data
3. Analyze patterns to determine root cause
4. Implement a fix and verify improvement
5. Create a reusable monitoring script to detect similar issues

The application contains multiple issues. Use the techniques covered in this module to systematically investigate. Document your approach and findings.

Final Project: Custom Observability Tool

Design and implement a production-grade custom observability tool using bpftrace. Your tool should address a specific monitoring or troubleshooting need that's not well-served by existing tools.

Requirements

- Target a specific performance or reliability problem
- Implement proper error handling and safety checks
- Include container awareness where appropriate
- Provide useful statistical aggregations or visualisations
- Document usage, output interpretation, and limitations
- Test across multiple kernel versions (5.10, 5.15, 6.1 if available)

Example Project Ideas

- Container-aware network latency profiler
- Memory fragmentation detector and alerting tool
- Cross-container request flow tracer
- Storage I/O anomaly detector
- Resource leak identifier for long-running containers

You'll present your tool to the class, explaining its design, implementation challenges, and demonstrating it in action on a realistic workload.

Best Practices Summary

1

Production Safety

- Test scripts thoroughly in non-production environments first
- Implement safeguards against unbounded output or memory usage
- Include timeouts and resource limits for all bpftrace scripts
- Validate on target kernel versions before deployment

2

Performance Impact

- Use sampling for high-frequency events
- Apply filters early in the script to reduce processing
- Limit map sizes through explicit cleanup or aggregation
- Consider batched output rather than per-event printing

3

Script Structure

- Follow consistent formatting and naming conventions
- Document assumptions and limitations inline
- Break complex logic into clear, focused sections
- Include version checks or fallbacks for kernel differences

4

Data Processing

- Use histograms and aggregations for large datasets
- Structure output for easy parsing by analysis tools
- Include enough context in output for offline analysis
- Consider outputting to structured formats (JSON, CSV)

Resources and Further Learning

Documentation & References

- **bpftrace Reference Guide**
- **BCC Reference Guide**
- **Linux Kernel BPF Documentation**
- **BPF Performance Tools (book by Brendan Gregg)**

Tools & Projects

- **bpftrace GitHub Repository**
- **BCC Tools Suite**
- **Cilium (eBPF-based networking)**
- **Tracee (security observability)**

Community & Support

- **IO Visor Mailing List**
- **eBPF Slack Channel**
- **LWN.net BPF Articles**

Advanced Topics

- CO-RE (Compile Once, Run Everywhere) for BPF
- Building custom BPF programs with libbpf
- eBPF for security monitoring and enforcement
- XDP (eXpress Data Path) for high-performance networking
- BPF LSM (Linux Security Module) for security policy

Q&A and Conclusion

Key Takeaways

- **Kernel Introspection:** bpftrace provides powerful capabilities for examining kernel data structures safely
- **Production Readiness:** Effective scripts require careful design for performance, safety, and reliability
- **Container Awareness:** Using cgroups and namespace understanding enables container-specific observability
- **Practical Application:** Real-world troubleshooting benefits from targeted, purpose-built instrumentation

Next Steps

Continue your learning journey by:

- Building a library of reusable bpftrace scripts for your environment
- Contributing to open-source eBPF tools and communities
- Exploring advanced BPF capabilities like CO-RE and XDP
- Integrating eBPF-based observability into your monitoring stack

Thank you for participating!

Questions?