Linux Observability and Tuning using bpftrace

An intensive hands-on training for advanced Linux performance analysis, debugging, and security monitoring using the powerful eBPF tracing language

Instructor: Chandrashekar Babu <training@chandrashekar.info>

https://www.chandrashekar.info/ | **https://www.slashprog.com/**

Module 2

bpftrace one-liners (contd.)
System-Wide Observability

Agenda: System-Wide Observability

1. **bpftrace one-liners (contd.)**
   Some more useful one-liner scripts

2. **Tracing Syscalls**
   Methods for observing syscall performance, counts, and failures

3. **Filesystem Operations**
   Tracing file access patterns, latency, and errors

4. **Process Events**
   Monitoring context switches, fork operations, and process lifecycle

5. **Kernel/Userspace Function Mapping**
   Techniques for correlating kernel and userspace activities

6. **Hands-On Labs**
   Real-world scenarios and practical applications

bpftrace Basics: Syntax Overview (recap)

```
// Basic structure of a bpftrace one-liner
probe [predicate] { actions }

// Example: Count system calls by process name
tracepoint:syscalls:sys_enter_*
{ @[comm] = count(); }

// Example: Probe with filtering
kprobe:vfs_read /pid == 181/
{ @bytes = sum(arg2); }

// Common built-in variables:
// pid, tid, comm, nsecs, elapsed, kstack, ustack, arg0-9
```

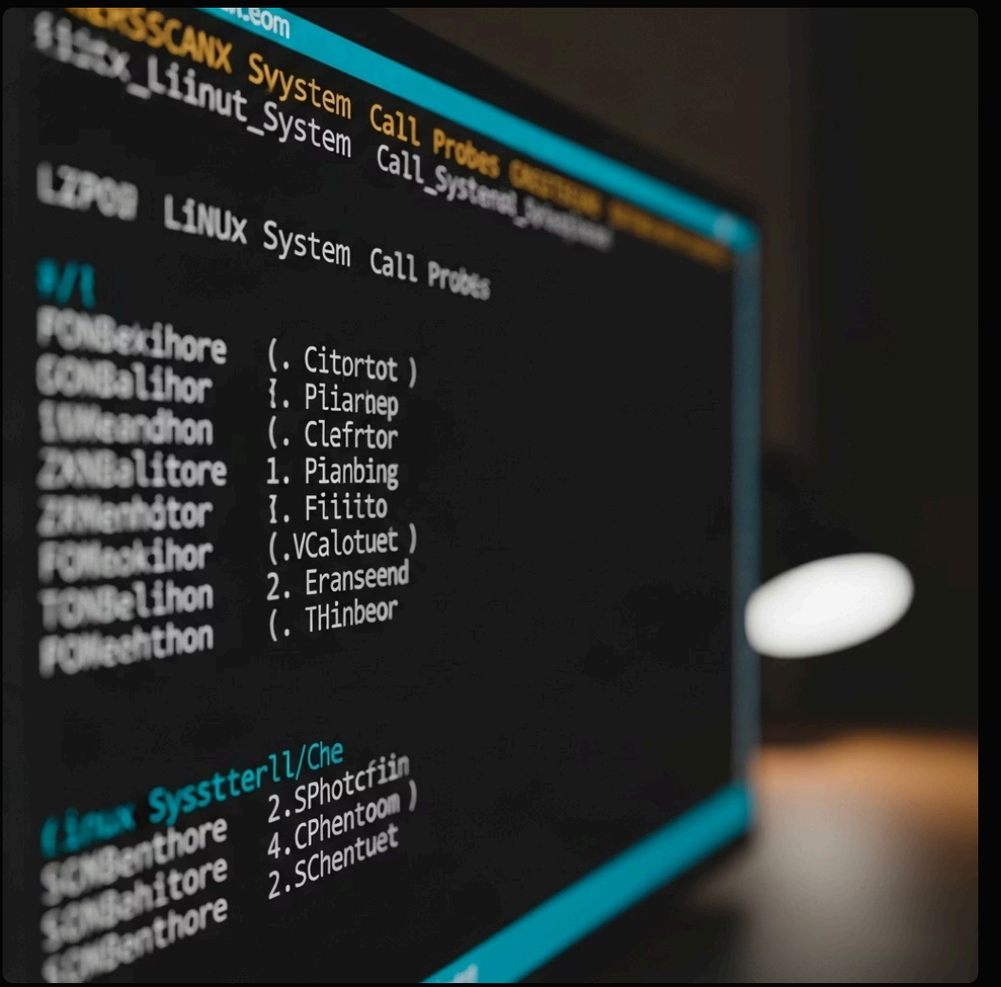
bpftrace probes connect to various sources: kprobes (kernel functions), uprobes (user functions), tracepoints, usdt probes, and more.

Listing Probes

```
bpftrace -l 'tracepoint:syscalls:sys_enter_*'
```

The **-l** option lists all available probes, and you can add a search term with wildcards to filter the results.

A probe is an instrumentation point for capturing event data. You can also pipe the output to grep for more advanced filtering.



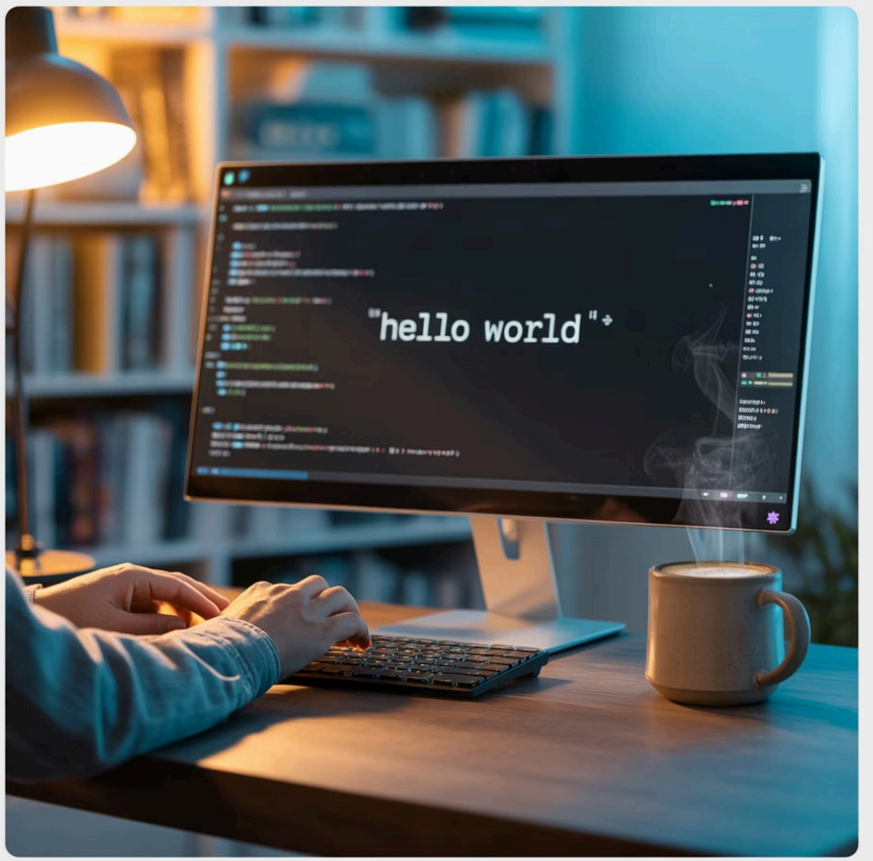
Probes are the foundation of bpftrace - they define *where* and *when* your tracing code will execute

Hello World

```
# bpftrace -e 'BEGIN { printf("hello world\n"); }'
Attaching 1 probe...
hello world
^C
```

Key Concepts:

- The **BEGIN** probe fires at the start of the program (similar to awk's BEGIN)
- Actions associated with probes are enclosed in curly braces { }
- This example calls printf() when the probe fires
- Press Ctrl-C to end the program

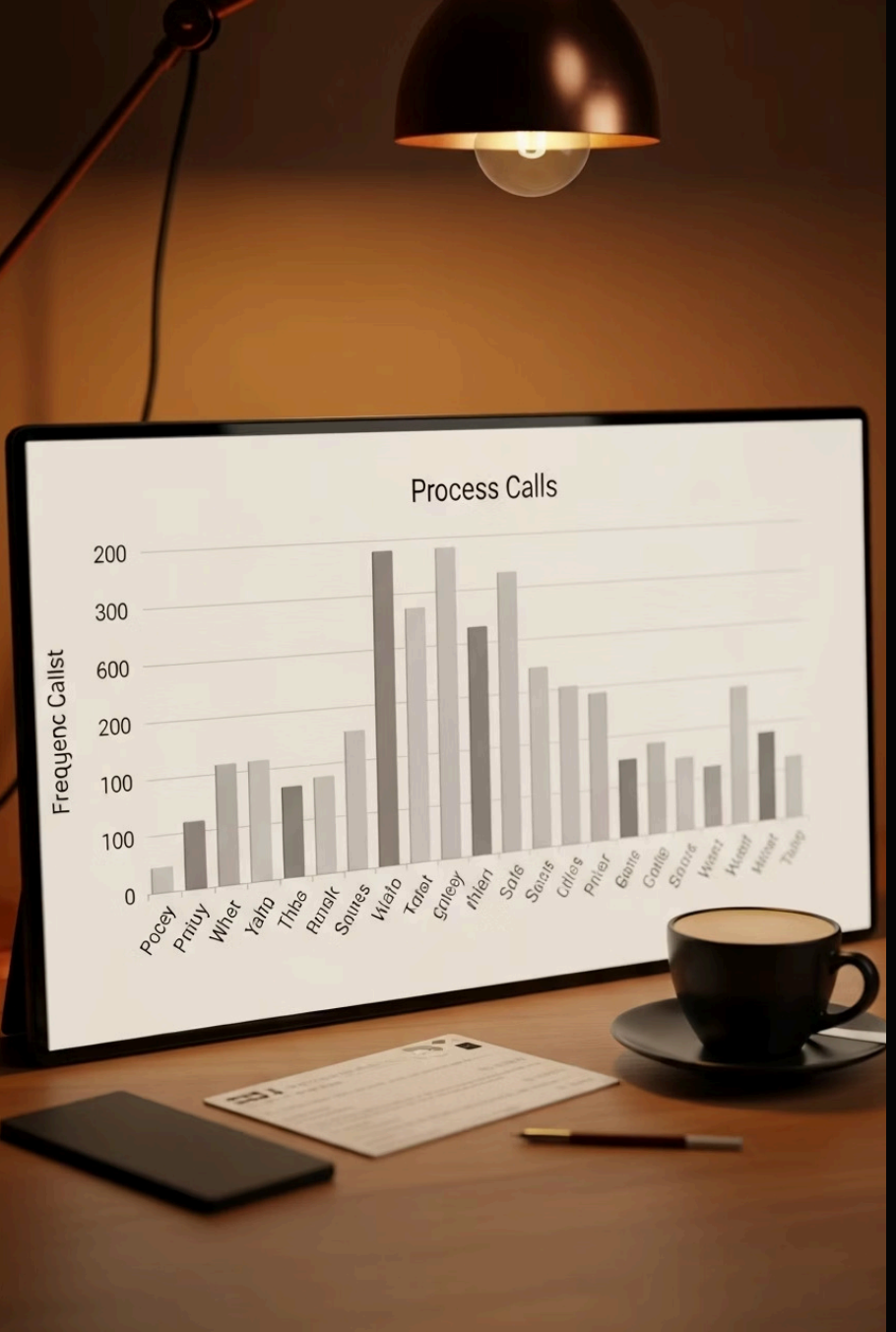


File Opens

```
# bpftrace -e 'tracepoint:syscalls:sys_enter_open* {
    printf("%s %s\n", comm, str(args.filename));
}'
```

This traces file opens as they happen, printing the process name and pathname.

- **tracepoint:syscalls:sys_enter_openat** is a kernel static tracing probe that fires when the openat() syscall begins
- **comm** is a builtin variable with the current process name
- **args** is a struct containing all tracepoint arguments
- **str()** converts a pointer to the string it points to



Syscall Counts By Process

```
bpftrace -e 'tracepoint:raw_syscalls:sys_enter {
    @[comm] = count();
}'
```

Maps

The **@** symbol denotes a special variable called a map that can store and summarize data in different ways

Keys

The brackets **[]** allow a key to be set for the map (like an associative array), in this case grouping by process name

Aggregation

count() is a map function that counts occurrences, creating a frequency count of syscalls by process

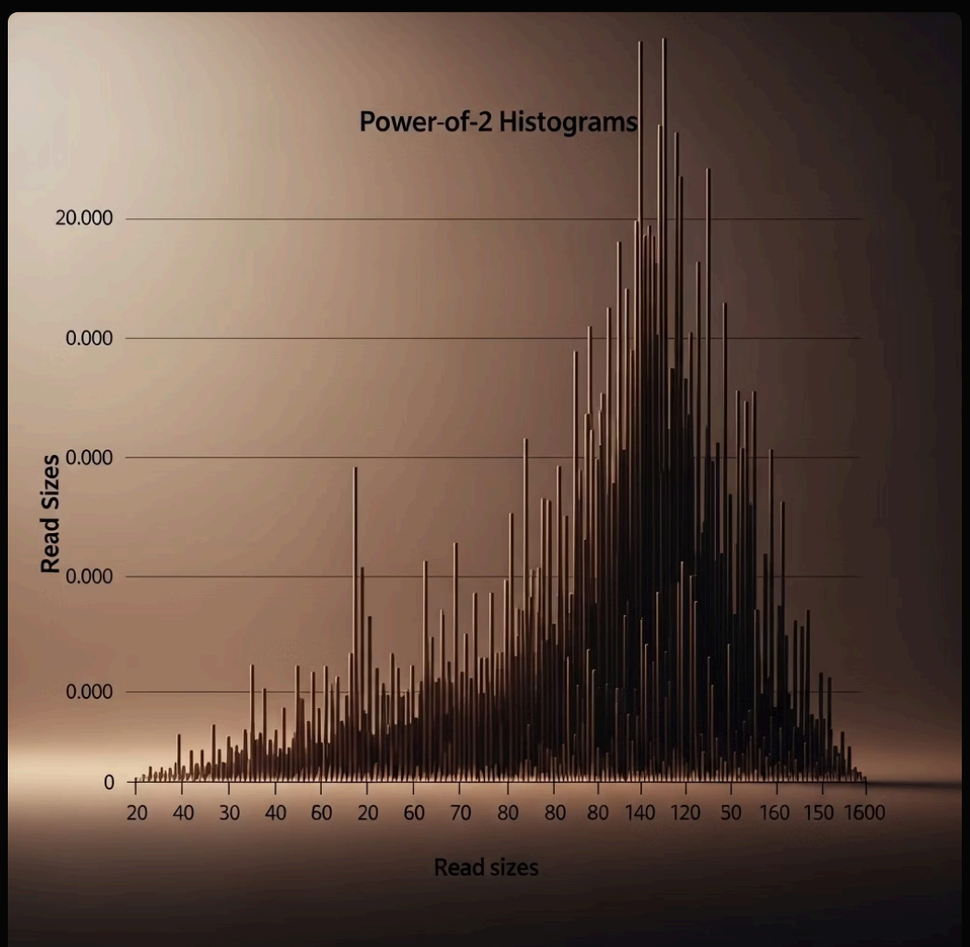
Maps are automatically printed when bpftrace ends (via Ctrl-C), showing which processes made the most syscalls.

Distribution of read() Bytes

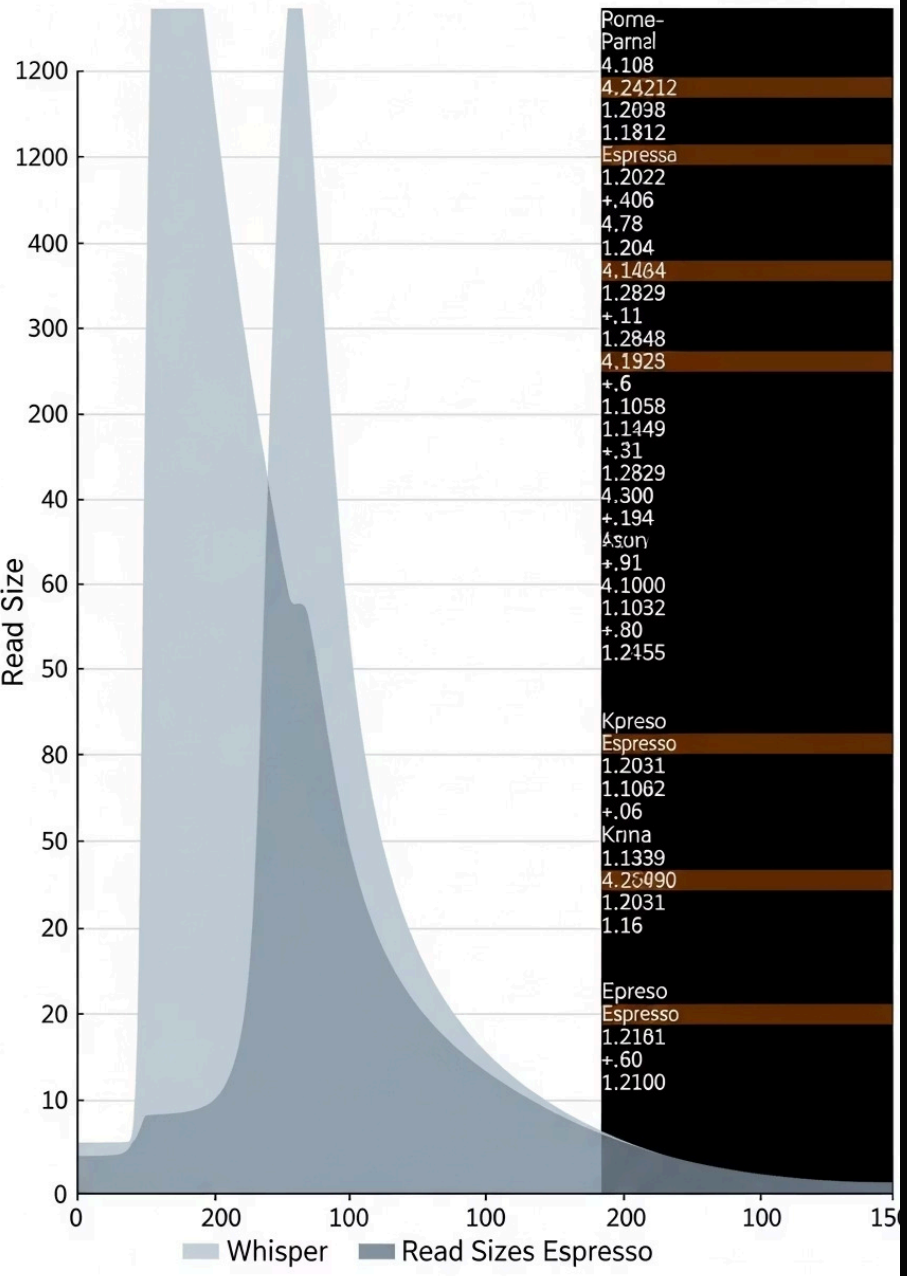
```
# bpftrace -e 'tracepoint:syscalls:sys_exit_read /pid == 18644/ {
    @bytes = hist(args->ret);
}'
```

Key Concepts:

- **/pid == 18644/** is a filter that only executes the action for the specified process ID
- **args→ret** is the return value of sys_read() - either -1 (error) or bytes read
- **@bytes** is a named map variable
- **hist()** creates a power-of-2 histogram of the values



The output shows how many reads occurred in each size range, with a visual representation



Kernel Dynamic Tracing of read() Bytes

```
# bpftrace -e 'kretprobe:vfs_read {
    @bytes = lhist(retval, 0, 2000, 200);
}'
```

This summarizes read() bytes as a linear histogram using kernel dynamic tracing.

- **kretprobe:vfs_read** instruments the return of the vfs_read() kernel function
- **lhist()** creates a linear histogram with arguments: value, min, max, step
- **retval** contains the return value (bytes read)

⚠ Kernel dynamic tracing is powerful but "unstable" - function names, arguments, and behaviors may change between kernel versions

Unlike tracepoints, kprobes/kretprobes require knowledge of kernel internals and may break with kernel updates

Timing read()s

```
# bpftrace -e '
kprobe:vfs_read {
    @start[tid] = nsecs;
}
kretprobe:vfs_read /@start[tid]/ {
    @ns[comm] = hist(nsecs - @start[tid]);
    delete(@start, tid);
}'
```

Multiple Probes

This script uses two probes: one at function entry and one at return

Timestamp Storage

@start[tid] stores the start time using thread ID as key, since each thread can only execute one syscall at a time

Duration Calculation

nsecs - @start[tid] calculates elapsed time in nanoseconds

Cleanup

delete(@start, tid) frees the variable to prevent memory leaks

The output shows how long read operations take for each process, helping identify performance bottlenecks.

Tracing Syscalls: Performance Latency

Let's measure how long specific syscalls take to execute:

```
# Measure read() syscall latency
bpftrace -e 'tracepoint:syscalls:sys_enter_read { @start[tid] = nsecs; }
            tracepoint:syscalls:sys_exit_read /@start[tid]/ {
                @latency_ns = hist(nsecs - @start[tid]);
                delete(@start[tid]);
            }'
```

This one-liner:

- Captures timestamp when read() syscall begins
- Calculates duration when read() completes
- Generates a histogram of latency distributions
- Cleans up the tracking hash map for completed syscalls

Count Process-Level Events

```
# bpftrace -e '
tracepoint:sched:sched* {
    @[probe] = count();
}
interval:s:5 {
    exit();
}'
```

This counts scheduler events for five seconds, then exits automatically and prints a summary.

- **tracepoint:sched:sched*** matches all scheduler tracepoints
- **@[probe]** counts occurrences by probe name
- **interval:s:5** fires once every 5 seconds on one CPU
- **exit()** terminates bpftrace

The sched category includes high-level scheduler and process events like fork, exec, and context switches.

Syscall Latency: More Examples

```
# Measure write() syscall latency by process
bpftrace -e 'tracepoint:syscalls:sys_enter_write { @start[tid] = nsecs; }
            tracepoint:syscalls:sys_exit_write /@start[tid]/ {
                @latency_ns[comm] = hist(nsecs - @start[tid]);
                delete(@start[tid]);
            }'

# Measure open() syscall latency with file path information
bpftrace -e 'tracepoint:syscalls:sys_enter_open {
                @start[tid] = nsecs;
                @filename[tid] = str(args->filename);
            }
            tracepoint:syscalls:sys_exit_open /@start[tid]/ {
                @latency_ns[@filename[tid]] = hist(nsecs - @start[tid]);
                delete(@start[tid]); delete(@filename[tid]);
            }'
```

These examples show how to add dimensions to your measurements (by process, by filename) for more targeted analysis.

Profiling Failed Syscalls

```
# List failed open syscalls with errno
bpftrace -e 'tracepoint:syscalls:sys_exit_open /args->ret < 0/ {
              @[comm, - args->ret] = count();
            }'

# Trace failed open syscalls with filenames and error messages
bpftrace -e 'BEGIN {
              // Define errno messages
              @errno[1] = "EPERM"; @errno[2] = "ENOENT";
              @errno[13] = "EACCES"; @errno[17] = "EEXIST";
            }
            tracepoint:syscalls:sys_enter_open {
              @fname[tid] = str(args->filename);
            }
            tracepoint:syscalls:sys_exit_open /args->ret < 0/ {
              printf("%s failed to open %s: %s\n",
                  comm, @fname[tid], @errno[- args->ret]);
              delete(@fname[tid]);
            }'
```

These scripts help identify permission issues, missing files, and other common error conditions.

Comprehensive Syscall Analysis Script

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

// syscall_analyzer.bt - Analyzes syscall patterns with detailed metrics

BEGIN {
 printf("Starting syscall analysis...\n");
 @start_time = nsecs;
}

tracepoint:syscalls:sys_enter_* {
 @syscalls_total++;
 @syscalls_by_proc[comm] += 1;
 @syscalls_by_type[probe] += 1;

 // Track start time for latency calculation
 @start[tid, probe] = nsecs;
}

tracepoint:syscalls:sys_exit_* /@start[tid, strcat("tracepoint:syscalls:sys_enter_",
 substr(probe, sizeof("tracepoint:syscalls:sys_exit_") - 1))]/ {
 $syscall = substr(probe, sizeof("tracepoint:syscalls:sys_exit_") - 1);
 $entry_probe = strcat("tracepoint:syscalls:sys_enter_", $syscall);
 $latency = nsecs - @start[tid, $entry_probe];

 // Record latency histogram
 @syscall_latency[$syscall] = hist($latency);

 // Track errors
 if (args->ret < 0) {
 @errors[$syscall, - args->ret] += 1;
 }

 delete(@start[tid, $entry_probe]);
}

interval:s:10 {
 printf("\n=== Syscall Statistics (10s interval) ===\n");
 printf("Total syscalls: %d\n", @syscalls_total);
 printf("\nTop 10 processes by syscall count:\n");
 print(@syscalls_by_proc, 10);
 printf("\nTop 10 syscall types:\n");
 print(@syscalls_by_type, 10);
 printf("\nError counts by syscall and errno:\n");
 print(@errors);
 clear(@syscalls_total);
 clear(@syscalls_by_proc);
 clear(@syscalls_by_type);
 clear(@errors);
}

END {
 printf("\n=== Final Latency Histograms ===\n");
 print(@syscall_latency);
 printf("\nTotal runtime: %.2f seconds\n", (nsecs - @start_time) / 1000000000);
}
```

Tracing Filesystem Operations

Filesystem operations are critical performance indicators for many applications. Let's see how to trace them with bpftrace.

```
# Count VFS (Virtual File System) calls by type
bpftrace -e 'kprobe:vfs_* { @[probe] = count(); }'

# Measure read sizes by process
bpftrace -e 'kprobe:vfs_read { @[comm] = hist(arg2); }'

# Track which files are being accessed most frequently
bpftrace -e 'kprobe:vfs_open {
 @[str(arg0)] = count();
 }'
```

VFS Operations Latency

```
# Measure latency of read operations
bpftrace -e 'kprobe:vfs_read { @start[tid] = nsecs; }
 kretprobe:vfs_read /@start[tid]/ {
 @read_latency_ns[comm] = hist(nsecs - @start[tid]);
 delete(@start[tid]);
 }'

# Measure latency of write operations
bpftrace -e 'kprobe:vfs_write { @start[tid] = nsecs; }
 kretprobe:vfs_write /@start[tid]/ {
 @write_latency_ns[comm] = hist(nsecs - @start[tid]);
 delete(@start[tid]);
 }'

# Measure total I/O time per process
bpftrace -e 'kprobe:vfs_read,kprobe:vfs_write { @start[tid] = nsecs; }
 kretprobe:vfs_read,kretprobe:vfs_write /@start[tid]/ {
 @io_time_ns[comm] += nsecs - @start[tid];
 delete(@start[tid]);
 }'
```

Filesystem Operations: vfsstat.bt

A comprehensive tool for measuring VFS operation statistics:

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

// vfsstat.bt - VFS operation statistics
//
// This script provides statistics on VFS (Virtual File System) operations,
// showing counts and latencies for the main VFS operations.

BEGIN {
 printf("Tracing VFS operations... Hit Ctrl-C to end.\n");
 // Track operation start time
}

kprobe:vfs_read,
kprobe:vfs_write,
kprobe:vfs_fsync,
kprobe:vfs_open,
kprobe:vfs_create {
 @start[tid] = nsecs;
 @counts[probe] += 1;
}

kretprobe:vfs_read /@start[tid]/ {
 @read_ns = hist(nsecs - @start[tid]);
 @totals["read"] += nsecs - @start[tid];
 delete(@start[tid]);
}

kretprobe:vfs_write /@start[tid]/ {
 @write_ns = hist(nsecs - @start[tid]);
 @totals["write"] += nsecs - @start[tid];
 delete(@start[tid]);
}

kretprobe:vfs_fsync /@start[tid]/ {
 @fsync_ns = hist(nsecs - @start[tid]);
 @totals["fsync"] += nsecs - @start[tid];
 delete(@start[tid]);
}

kretprobe:vfs_open /@start[tid]/ {
 @open_ns = hist(nsecs - @start[tid]);
 @totals["open"] += nsecs - @start[tid];
 delete(@start[tid]);
}

kretprobe:vfs_create /@start[tid]/ {
 @create_ns = hist(nsecs - @start[tid]);
 @totals["create"] += nsecs - @start[tid];
 delete(@start[tid]);
}

interval:s:5 {
 time("%H:%M:%S\n");
 print(@counts);
 printf("\nOperation totals (nanoseconds):\n");
 print(@totals);
 printf("\nLatency histograms:\n");
 printf("VFS read latency (ns):\n");
 print(@read_ns);
 clear(@counts);
 clear(@totals);
 clear(@read_ns);
 clear(@write_ns);
 clear(@fsync_ns);
 clear(@open_ns);
 clear(@create_ns);
}

END {
 printf("Finished tracing VFS operations.\n");
}
```

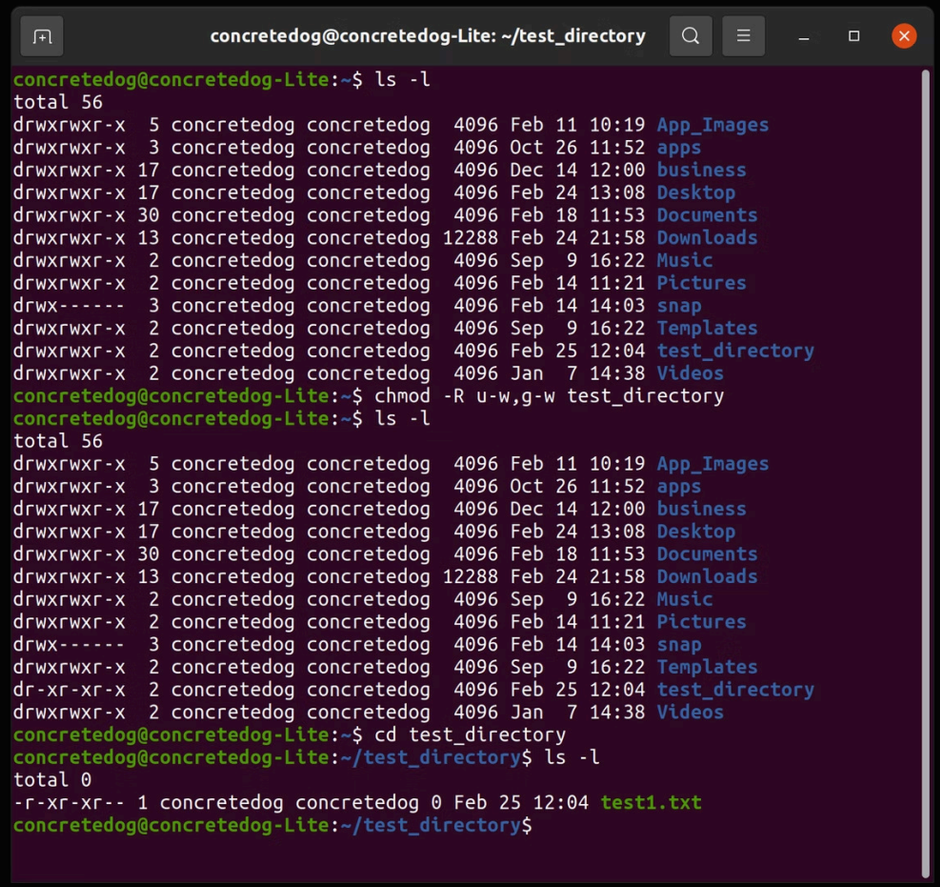
Finding Slow File Access Patterns

```
# Trace files with high read latency
bpftrace -e 'kprobe:vfs_read {
 @path[tid] = arg0;
 @start[tid] = nsecs;
 }
 kretprobe:vfs_read /@start[tid] && @path[tid]/ {
 $duration = nsecs - @start[tid];
 // Only show reads that took > 1ms
 if ($duration > 1000000) {
 printf("Slow read (%d ms): %s by %s\n",
 $duration / 1000000, @path[tid], comm);
 }
 delete(@path[tid]);
 delete(@start[tid]);
 }'

# Track amount of data read/written per file (requires helper function)
bpftrace -e '
 // Helper function to retrieve dentry name (simplified)
 function get_dentry_name(struct dentry *dentry) {
 return str(dentry->d_name.name);
 }

 kprobe:vfs_read {
 $file = (struct file *)arg0;
 $dentry = $file->f_path.dentry;
 @bytes_read[get_dentry_name($dentry), comm] += arg2;
 }

 kprobe:vfs_write {
 $file = (struct file *)arg0;
 $dentry = $file->f_path.dentry;
 @bytes_written[get_dentry_name($dentry), comm] += arg2;
 }'
```

Detecting Failed File Accesses

Failed file access attempts can indicate permission issues, misconfigurations, or potential security problems.

```
# Track failed open() calls with filename and error
bpftrace -e '
    tracepoint:syscalls:sys_enter_open {
        @fname[tid] = str(args->filename);
    }
    tracepoint:syscalls:sys_exit_open /args->ret < 0/ {
        printf("%s (pid %d) failed to open %s: errno %d\n",
               comm, pid, @fname[tid], -args->ret);
        delete(@fname[tid]);
    }'
```



Common reasons for failed file access:

- Permission denied (EACCES - errno 13)
- File not found (ENOENT - errno 2)
- File exists (EEXIST - errno 17)
- Too many open files (EMFILE - errno 24)

Failed File Access Detection Script

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

// failedaccess.bt - Track all failed file access attempts

BEGIN {
 printf("Tracing failed file access attempts... Hit Ctrl-C to end.\n\n");

 // Define common errno values and their meanings
 @errors[1] = "EPERM: Operation not permitted";
 @errors[2] = "ENOENT: No such file or directory";
 @errors[3] = "ESRCH: No such process";
 @errors[4] = "EINTR: Interrupted system call";
 @errors[5] = "EIO: I/O error";
 @errors[6] = "ENXIO: No such device or address";
 @errors[13] = "EACCES: Permission denied";
 @errors[17] = "EEXIST: File exists";
 @errors[24] = "EMFILE: Too many open files";
}

// Track filenames for opens
tracepoint:syscalls:sys_enter_open,
tracepoint:syscalls:sys_enter_openat {
 @filename[tid] = str(args->filename);
}

// Track failed opens
tracepoint:syscalls:sys_exit_open,
tracepoint:syscalls:sys_exit_openat /@filename[tid] && args->ret < 0/ {
 $errno = - args->ret;
 $error_msg = @errors[$errno] ? @errors[$errno] : str($errno);

 printf("%-16s %-6d %-6d FAILED OPEN: %s (%s)\n",
 comm, pid, tid, @filename[tid], $error_msg);

 // Collect statistics
 @fails_by_process[comm] += 1;
 @fails_by_file[@filename[tid]] += 1;
 @fails_by_error[$error_msg] += 1;

 delete(@filename[tid]);
}

// Track failed reads
kretprobe:vfs_read /retval < 0/ {
 $file = (struct file *)arg0;
 $dentry = $file->f_path.dentry;
 $filename = str($dentry->d_name.name);
 $errno = - retval;
 $error_msg = @errors[$errno] ? @errors[$errno] : str($errno);

 printf("%-16s %-6d %-6d FAILED READ: %s (%s)\n",
 comm, pid, tid, $filename, $error_msg);

 @fails_by_process[comm] += 1;
 @fails_by_file[$filename] += 1;
 @fails_by_error[$error_msg] += 1;
}

// Print summary on exit
END {
 printf("\n=== Summary ===\n");
 printf("\nFailures by process:\n");
 print(@fails_by_process);

 printf("\nFailures by file:\n");
 print(@fails_by_file);

 printf("\nFailures by error:\n");
 print(@fails_by_error);
}
```

Profile On-CPU Kernel Stacks

```
# bpftrace -e 'profile:hz:99 { @[kstack] = count(); }'
```

Key Concepts:

- **profile:hz:99** samples all CPUs at 99 Hertz
- 99Hz is frequent enough to capture execution patterns without significantly impacting performance
- **kstack** returns the kernel stack trace
- Using the stack trace as a map key creates a frequency count of execution paths

The output is ideal for visualizing as a flame graph to understand kernel execution patterns

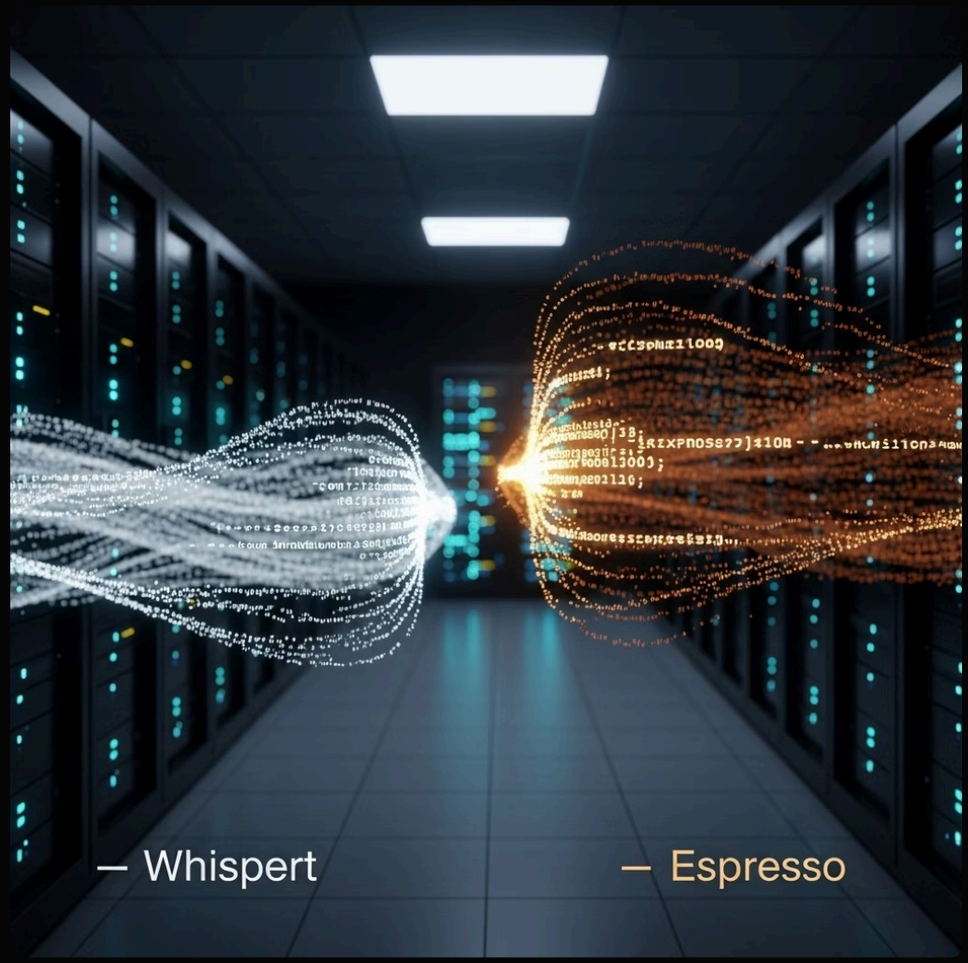
Scheduler Tracing

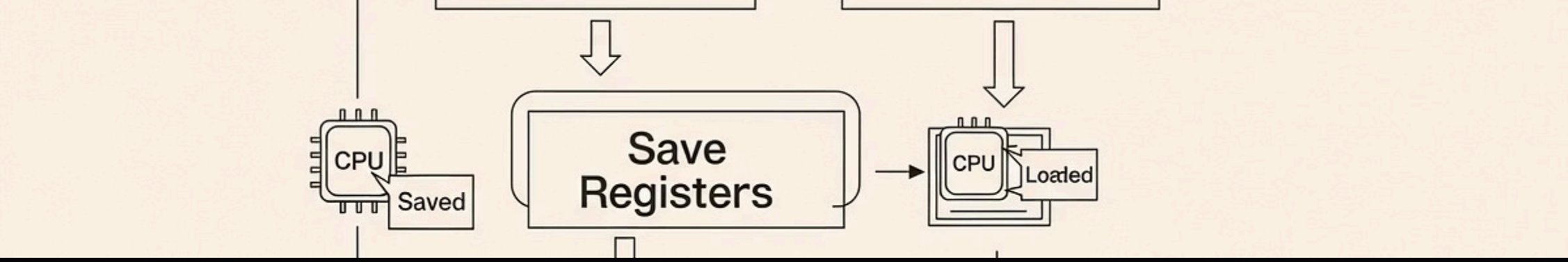
```
# bpftrace -e 'tracepoint:sched:sched_switch { @[kstack] = count(); }'
```

This counts stack traces that led to context switching (off-CPU) events.

- **sched_switch** fires when a thread leaves CPU
- These are blocking events: waiting on I/O, timers, paging/swapping, or locks
- The stack trace shows what code path led to the context switch

Context is important: sched_switch fires in thread context, so the stack refers to the thread that is leaving the CPU





Process Events: Context Switches

Context switches can be a major source of performance overhead. Let's trace them:

```
# Count context switches by process
bpftrace -e 'tracepoint:sched:sched_switch { @[comm] = count(); }'

# Measure time spent context switching
bpftrace -e 'tracepoint:sched:sched_switch {
 @start[cpu] = nsecs;
 }
 tracepoint:sched:sched_switch /@start[cpu]/ {
 @switch_time[cpu] = sum(nsecs - @start[cpu]);
 @start[cpu] = nsecs;
 }'

# Identify processes that cause frequent context switches
bpftrace -e 'tracepoint:sched:sched_switch {
 @from[args->prev_comm] = count();
 @to[args->next_comm] = count();
 }'
```

Advanced Context Switch Analysis

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

// switchanalysis.bt - Analyze context switching patterns

BEGIN {
 printf("Analyzing context switches... Hit Ctrl-C to end.\n");
}

// Track when a thread is scheduled off CPU
tracepoint:sched:sched_switch {
 // Record when the thread went off-CPU
 $pid = args->prev_pid;
 $comm = args->prev_comm;
@last_off_time[$pid, $comm] = nsecs;

// Calculate time spent off-CPU for the thread that's now scheduled on
$new_pid = args->next_pid;
$new_comm = args->next_comm;
$last_time = @last_off_time[$new_pid, $new_comm];
if ($last_time != 0) {
$off_time = nsecs - $last_time;
@off_cpu_time_us[$new_comm, $new_pid] = hist($off_time / 1000);
delete(@last_off_time[$new_pid, $new_comm]);
}

// Track how often processes are switched
@switch_count[$comm, "switched out"] += 1;
@switch_count[$new_comm, "switched in"] += 1;

// Track common pairs (what often follows what)
@pairs[$comm, $new_comm] += 1;
}

// Track run queue latency (time spent waiting to run)
tracepoint:sched:sched_wakeup {
 @wakeup_time[args->pid, args->comm] = nsecs;
}

tracepoint:sched:sched_switch /@wakeup_time[args->next_pid, args->next_comm]/ {
 $latency = nsecs - @wakeup_time[args->next_pid, args->next_comm];
 @runq_latency_us[args->next_comm] = hist($latency / 1000);
 delete(@wakeup_time[args->next_pid, args->next_comm]);
}

interval:s:10 {
 printf("\n=== Context Switch Statistics ===\n");
 printf("\nTop process switch counts:\n");
 print(@switch_count, 10);

 printf("\nCommon process pairs (what follows what):\n");
 print(@pairs, 10);

 clear(@switch_count);
 clear(@pairs);
}

END {
 printf("\n=== Final Statistics ===\n");
 printf("\nTime spent off-CPU by process (microseconds):\n");
 print(@off_cpu_time_us);

 printf("\nRun queue latency by process (microseconds):\n");
 print(@runq_latency_us);
}
```

Process Events: fork() Operations

Tracking process creation provides insights into application behavior and system load:

```
# Count fork() calls by process
bpftrace -e 'tracepoint:syscalls:sys_enter_fork { @[comm] = count(); }'

# Track parent-child relationships
bpftrace -e 'tracepoint:sched:sched_process_fork {
 printf("%s (pid %d) forked %s (pid %d)\n",
 args->parent_comm, args->parent_pid,
 args->child_comm, args->child_pid);
 }'

# Measure time between fork() and exec()
bpftrace -e 'tracepoint:sched:sched_process_fork {
 @start[args->child_pid] = nsecs;
 }
 tracepoint:sched:sched_process_exec /@start[pid]/ {
 @fork_to_exec = hist(nsecs - @start[pid]);
 delete(@start[pid]);
 }'
```

execsnoop.bt: Tracking Process Execution

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

// execsnoop.bt - Track process execution
// Inspired by Brendan Gregg's execsnoop

BEGIN {
 printf("%-10s %-6s %-6s %-6s %s\n", "TIME(ms)", "PID", "PPID", "RET", "COMMAND");
}

tracepoint:syscalls:sys_enter_execve {
 // Remember the executable name for this PID
 @exec[pid] = str(args->filename);
 @args[pid] = "";

 // Collect command-line arguments
 $argp = (char **)args->argv;
 if ($argp != 0) {
 $arg = (char *)$argp;
 $i = 1;
 while (($arg = (char *)$argp[$i])) {
 @args[pid] = strcat(@args[pid], " ");
 @args[pid] = strcat(@args[pid], str($arg));
 $i++;
 }
 }

 // Remember parent PID
 @ppid[pid] = pid;
}

tracepoint:syscalls:sys_exit_execve {
 $now = nsecs / 1000000; // Convert to milliseconds
 printf("%-10u %-6d %-6d %-6d %s%s\n",
 $now, pid, @ppid[pid], args->ret,
 @exec[pid] ? @exec[pid] : "?",
 @args[pid] ? @args[pid] : "");

 // Clean up our arrays
 delete(@exec[pid]);
 delete(@args[pid]);
 delete(@ppid[pid]);
}

// Handle fork events to track parent-child relationships
tracepoint:sched:sched_process_fork {
 @ppid[args->child_pid] = args->parent_pid;
}

END {
 clear(@exec);
 clear(@args);
 clear(@ppid);
}
```

Performance Bottleneck Detection

Identify which syscalls are performance bottlenecks for a specific process:

```
# Measure time spent in syscalls for a specific process
bpftrace -e 'tracepoint:syscalls:sys_enter_* /comm == "nginx"/ {
                @start[tid, probe] = nsecs;
            }
            tracepoint:syscalls:sys_exit_* /comm == "nginx" &&
                @start[tid, strcat("tracepoint:syscalls:sys_enter_",
                substr(probe, sizeof("tracepoint:syscalls:sys_exit_") - 1))]/
{
                $syscall = substr(probe,
                        sizeof("tracepoint:syscalls:sys_exit_") - 1);
                $entry = strcat("tracepoint:syscalls:sys_enter_", $syscall);
                @time[$syscall] = sum(nsecs - @start[tid, $entry]);
                delete(@start[tid, $entry]);
            }'
```

Profile on-CPU time with stacks:

```
# Sample kernel stacks for a process
bpftrace -e 'profile:hz:99 /comm == "nginx"/ {
 @[kstack] = count();
 }'

# Sample user stacks for a process
bpftrace -e 'profile:hz:99 /comm == "nginx"/ {
 @[ustack] = count();
 }'

# Combined kernel+user stacks
bpftrace -e 'profile:hz:99 /comm == "nginx"/ {
 @[kstack, ustack] = count();
 }'
```

Syscall Bottleneck Analyzer

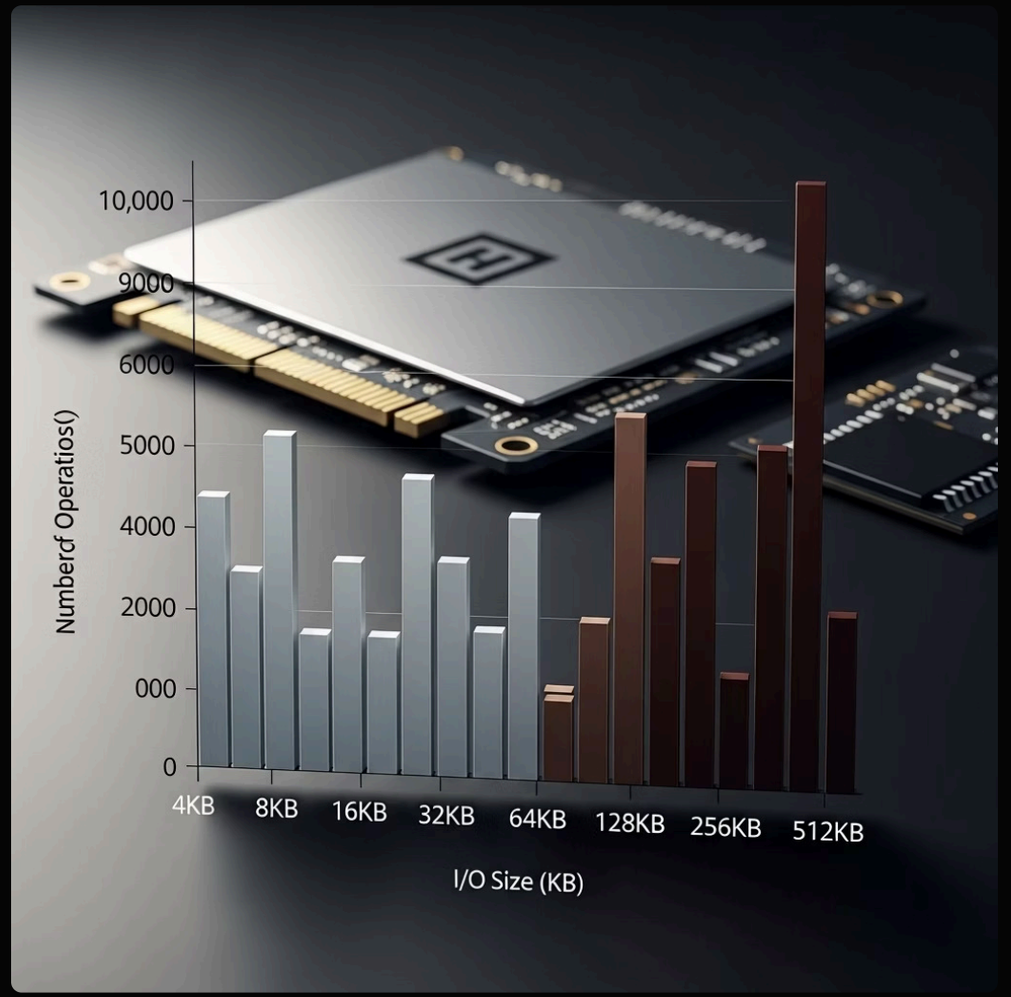
```
#!/usr/bin/env bpftrace

// syscall_bottleneck.bt - Find syscall bottlenecks for a specific process
// Usage: bpftrace syscall_bottleneck.bt [process_name]

BEGIN {
 $target_process = str($1);
 printf("Analyzing syscall bottlenecks for process: %s\n", $target_process);
 printf("Hit Ctrl-C to end and display summary.\n");

 @start_time = nsecs;
}

// Record syscall entry
tracepoint:syscalls:sys_enter_* /comm == str($1)/ {
 $syscall = substr(probe, sizeof("tracepoint:syscalls:sys_enter_") - 1);
 @start[tid, $syscall] = nsecs;
 @calls[$syscall] += 1;
}

// Record syscall exit and calculate duration
tracepoint:syscalls:sys_exit_* /comm == str($1) &&
 @start[tid, substr(probe,
 sizeof("tracepoint:syscalls:sys_exit_") - 1)]/ {
 $syscall = substr(probe, sizeof("tracepoint:syscalls:sys_exit_") - 1);
 $duration = nsecs - @start[tid, $syscall];

 @total_time[$syscall] += $duration;
 @time_hist[$syscall] = hist($duration);

 // Track errors
 if (args->ret < 0) {
 @errors[$syscall, -args->ret] += 1;
 }

 delete(@start[tid, $syscall]);
}

// Periodically take kernel and user stacks to correlate with syscalls
profile:hz:99 /comm == str($1)/ {
 @kstacks[kstack] += 1;
 @ustacks[ustack] += 1;
}

END {
 $runtime_ns = nsecs - @start_time;
 $runtime_s = $runtime_ns / 1000000000;

 printf("\n=== Syscall Statistics for %s (Runtime: %.2f seconds) ===\n",
 str($1), $runtime_s);

 printf("\nTop syscalls by total time:\n");
 // Calculate percentages and print sorted results
 $syscalls = print(@total_time, 10);

 printf("\nTop syscalls by call count:\n");
 print(@calls, 10);

 printf("\nSyscall error counts:\n");
 print(@errors);

 printf("\nSyscall latency histograms (nanoseconds):\n");
 print(@time_hist);

 printf("\nTop kernel stacks:\n");
 print(@kstacks, 5);

 printf("\nTop user stacks:\n");
 print(@ustacks, 5);
}
```

Block I/O Tracing

```
# bpftrace -e 'tracepoint:block:block_rq_issue { @ = hist(args.bytes); }'
```

This creates a histogram of block I/O requests by size in bytes.

- **tracepoint:block** category traces storage events
- **block_rq_issue** fires when an I/O is issued to the device
- **args.bytes** shows the size of the I/O request
- The histogram shows the distribution of I/O sizes



Understanding I/O size patterns helps optimize storage performance and identify inefficient access patterns

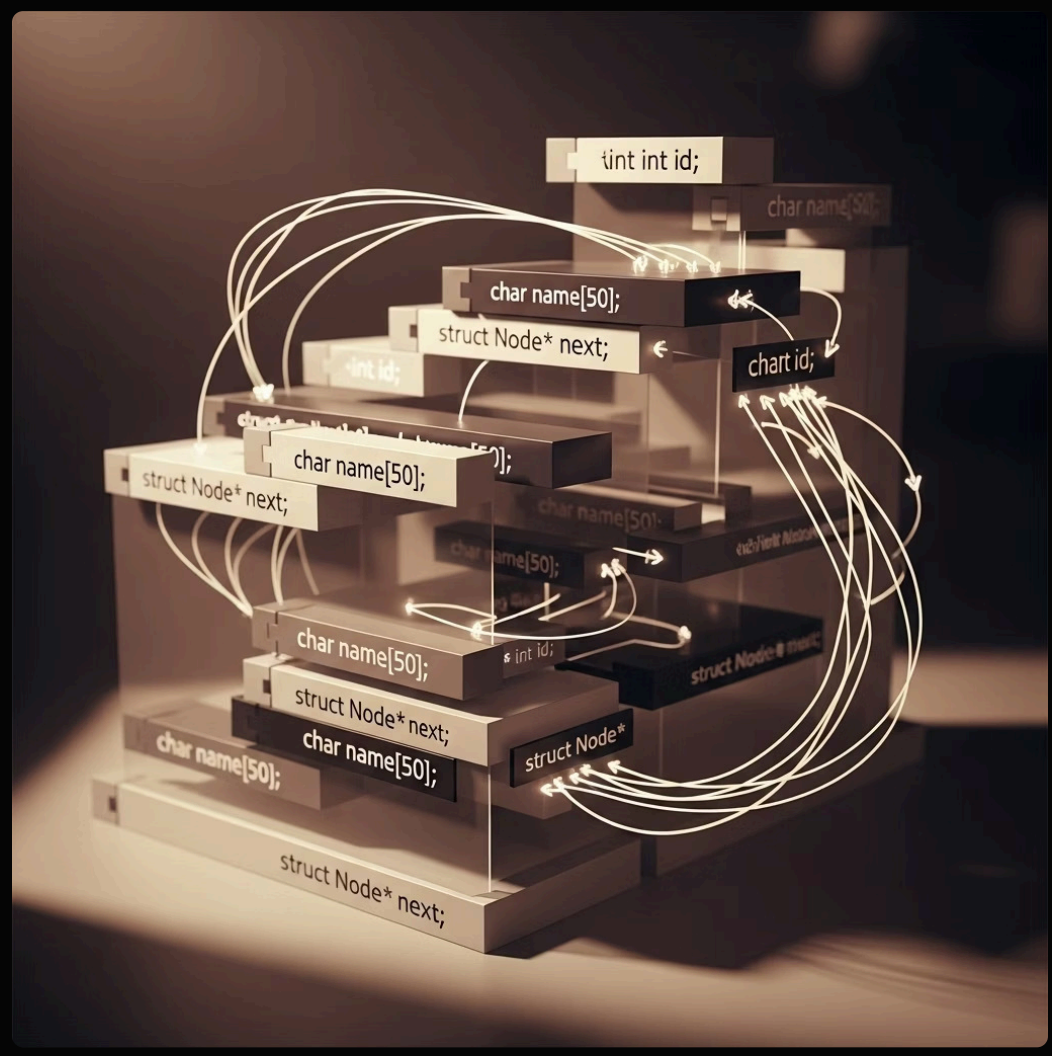
Kernel Struct Tracing

```
# cat path.bt
#ifndef BPFTRACE_HAVE_BTF
#include
#include
#endif

kprobe:vfs_open {
    printf("open path: %s\n",
        str(((struct path *)arg0)->dentry->d_name.name));
}
```

This uses kernel dynamic tracing to access struct members from the vfs_open() function.

- **arg0** contains the first function argument
- The code casts arg0 to a struct path pointer and accesses nested members
- Include statements are needed on systems without BTF (BPF Type Format) data

With BTF-enabled kernels, all kernel structs are automatically available without includes



Key bpftrace Concepts

Probe Types

Different ways to instrument the system:

- **tracepoint:** Stable kernel static tracing points
- **kprobe/kretprobe:** Dynamic kernel function tracing
- **profile:** Timer-based sampling
- **interval:** Periodic events
- **BEGIN/END:** Program start/end events

Builtin Variables

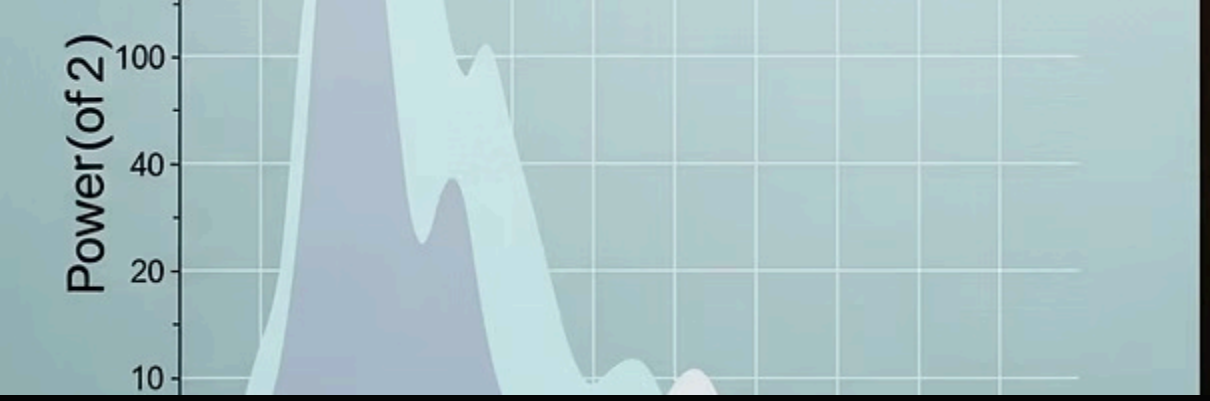
Contextual information available in probes:

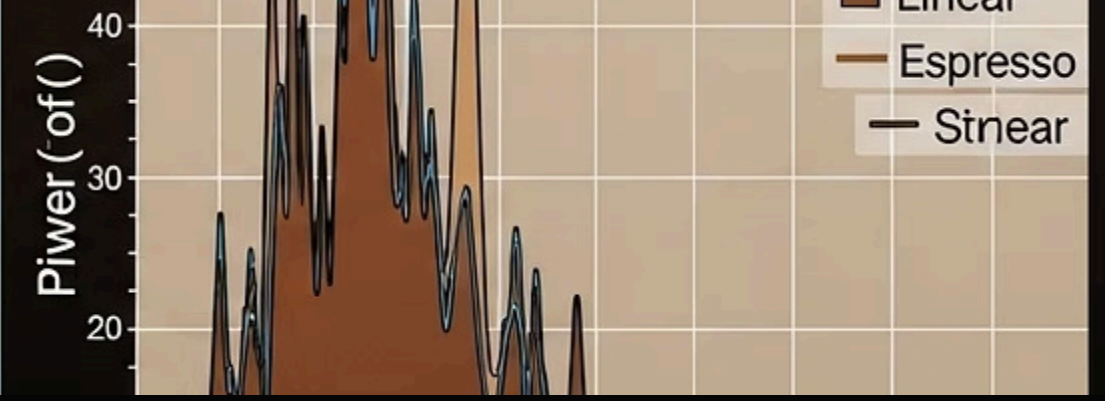
- **pid/tid:** Process/thread IDs
- **comm:** Process name
- **nsecs:** Timestamp in nanoseconds
- **kstack/ustack:** Kernel/user stack traces
- **arg0...argN:** Function arguments
- **retval:** Return value (kretprobe)

Maps & Aggregations

Data storage and analysis:

- **@name[key]:** Store/aggregate values
- **count()/sum()/avg():** Basic statistics
- **min()/max():** Value ranges
- **hist()/lhist():** Histograms
- Maps print automatically at program end





Histograms: Visualizing Distributions

Power-of-2 Histogram (hist())

```
@bytes = hist(args.ret);
```

Creates buckets in powers of 2: [0,1], [2,4), [4,8), etc.

Ideal for values that vary by orders of magnitude (like bytes read)

Linear Histogram (lhist())

```
@latency = lhist(value, 0, 100, 10);
```

Creates evenly-spaced buckets: [0,10), [10,20), etc.

Better for values in a known, limited range (like percentages)

Histograms help identify patterns, outliers, and multi-modal distributions in your data that simple averages would miss

Custom Metrics with Histograms

Histograms provide powerful visualizations of data distributions:

```
# Basic histogram of read sizes
bpftrace -e 'kprobe:vfs_read {
        @bytes = hist(arg2);
      }'

# Output:
# @bytes:
# [0, 1]          12 |                                  |
# [2, 4)          25 |@                                 |
# [4, 8)         121 |@@@@                              |
# [8, 16)       1214
|@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@|
# [16, 32)       508 |@@@@@@@@@@@@@@@@                  |
# [32, 64)       271 |@@@@@@@@@                         |
# [64, 128)       38 |@                                 |
```

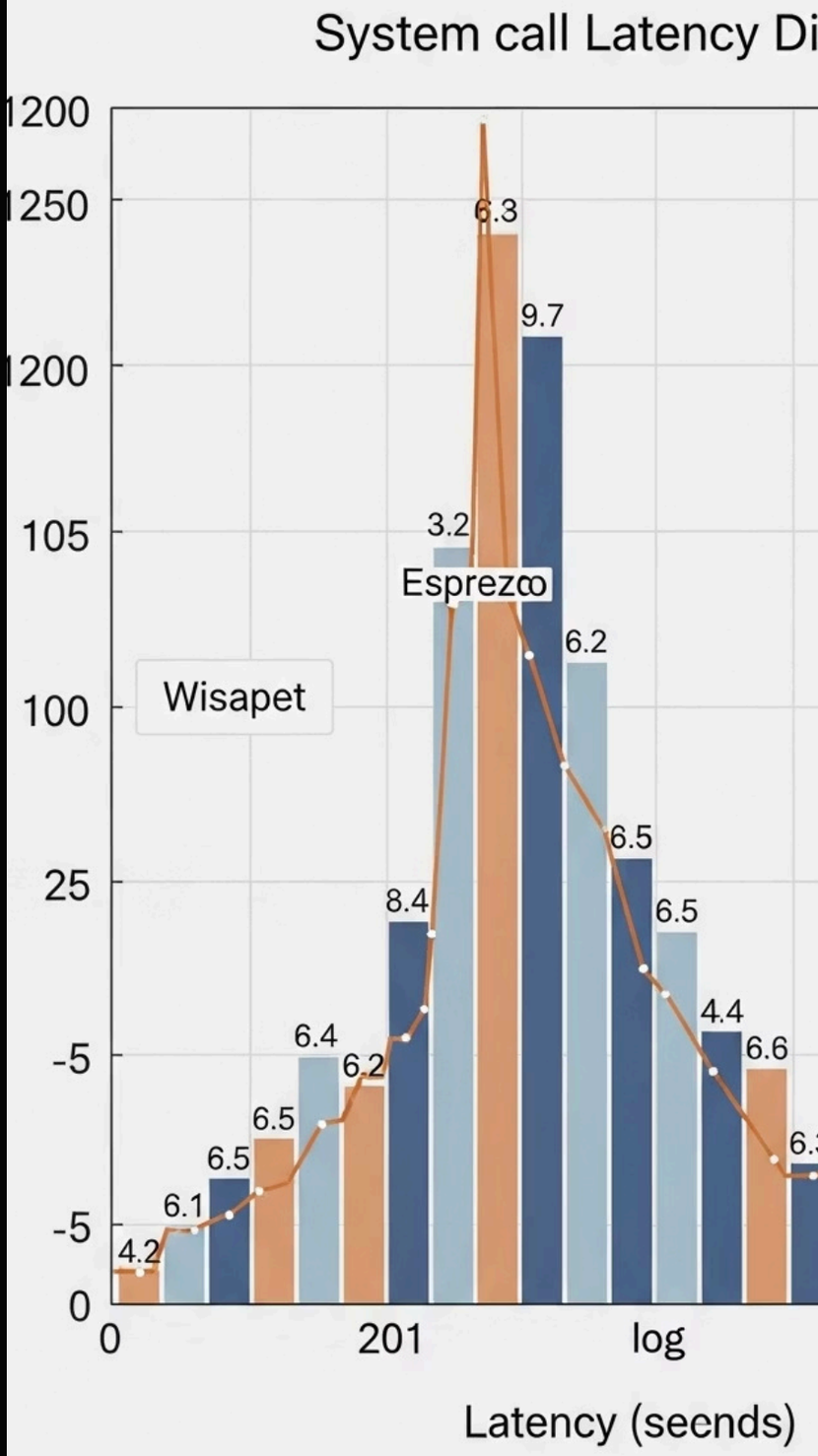
Histogram types:

- **hist()** - power-of-2 histogram
- **lhist()** - linear histogram with custom buckets

```
# Linear histogram with custom buckets
bpftrace -e 'kprobe:vfs_read {
 @bytes = lhist(arg2, 0, 1000, 100);
 }'

# Multi-dimensional histogram
bpftrace -e 'kprobe:vfs_read {
 @bytes[comm] = hist(arg2);
 }'
```

Histogram Examples

```
# Syscall latency histogram by process
bpftrace -e '
 tracepoint:syscalls:sys_enter_read {
 @start[tid] = nsecs;
 }

 tracepoint:syscalls:sys_exit_read /@start[tid]/ {
 @read_latency_ns[comm] = hist(nsecs - @start[tid]);
 delete(@start[tid]);
 }'

# Linear histogram of process lifetimes (0-60s in 5s buckets)
bpftrace -e '
 tracepoint:sched:sched_process_fork {
 @birth[args->child_pid] = nsecs;
 }

 tracepoint:sched:sched_process_exit /@birth[args->pid]/ {
 $lifetime_s = (nsecs - @birth[args->pid]) / 1000000000;
 @process_lifetime_s = lhist($lifetime_s, 0, 60, 5);
 delete(@birth[args->pid]);
 }'

# Memory allocation size distribution
bpftrace -e '
 kprobe:kmem_cache_alloc {
 @alloc_bytes = hist(arg1);
 }'
```



Custom Histogram Script

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

// disklatency.bt - Disk I/O latency histograms

BEGIN {
 printf("Measuring disk I/O latency... Hit Ctrl-C to end.\n");
 printf("Collecting data across different dimensions...\n");
}

// Block I/O request start
kprobe:blk_start_request,
kprobe:blk_mq_start_request {
 // Save request start time
 @start[arg0] = nsecs;
}

// Block I/O completion
kprobe:blk_account_io_done /@start[arg0]/ {
 $duration = nsecs - @start[arg0];
 $disk = ((struct request *)arg0)->rq_disk;

 // Extract I/O information
 $device = $disk->disk_name;
 $major = $disk->major;
 $minor = $disk->minor;

 // Read/write direction
 $read = ((struct request *)arg0)->cmd_flags & 1; // REQ_OP_READ
 $op_type = $read ? "read" : "write";

 // Request size in 512-byte sectors
 $sectors = ((struct request *)arg0)->__sector;
 $bytes = $sectors * 512;

 // Store data in multidimensional histograms
 @total_latency_us = hist($duration / 1000);
 @latency_by_device_us[$device] = hist($duration / 1000);
 @latency_by_operation_us[$op_type] = hist($duration / 1000);
 @latency_by_size_us[hist_to_size($bytes)] = hist($duration / 1000);

 // Average latency by device and operation
 @avg_latency_us[$device, $op_type] = avg($duration / 1000);

 delete(@start[arg0]);
}

// Helper functions to categorize sizes
function hist_to_size(bytes) {
 if (bytes < 4096) { return "0-4K"; }
 if (bytes < 16384) { return "4K-16K"; }
 if (bytes < 65536) { return "16K-64K"; }
 if (bytes < 262144) { return "64K-256K"; }
 return ">=256K";
}

interval:s:10 {
 printf("\nAverage latency by device and operation (microseconds):\n");
 print(@avg_latency_us);
 clear(@avg_latency_us);
}

END {
 printf("\n=== I/O Latency Summary ===\n");

 printf("\nOverall I/O latency distribution (microseconds):\n");
 print(@total_latency_us);

 printf("\nLatency by device (microseconds):\n");
 print(@latency_by_device_us);

 printf("\nLatency by operation type (microseconds):\n");
 print(@latency_by_operation_us);

 printf("\nLatency by request size (microseconds):\n");
 print(@latency_by_size_us);
}
```

Advanced Filtering Techniques

Process Filters

```
/pid == 1234/
/comm == "nginx"/
```

Target specific processes or applications

Value Filters

```
/args.ret < 0/
/args.flags & O_CREAT/
```

Filter based on arguments, return values, or bit flags

Stack Filters

```
/kstack contains "tcp_sendmsg"/
```

Match based on function names in the stack trace

Compound Filters

```
/pid == 1234 && args.ret < 0/
/comm == "nginx" || comm == "httpd"/
```

Combine conditions with AND (&&) and OR (||) operators

Effective filtering is crucial for focusing on relevant events and reducing overhead in production environments

Best Practices

Performance Considerations

- Use filters early to reduce processing overhead
- Avoid string operations in high-frequency probes
- Be cautious with stack traces in hot paths
- Test on non-production systems first
- Set reasonable sampling rates (e.g., 99Hz not 9999Hz)

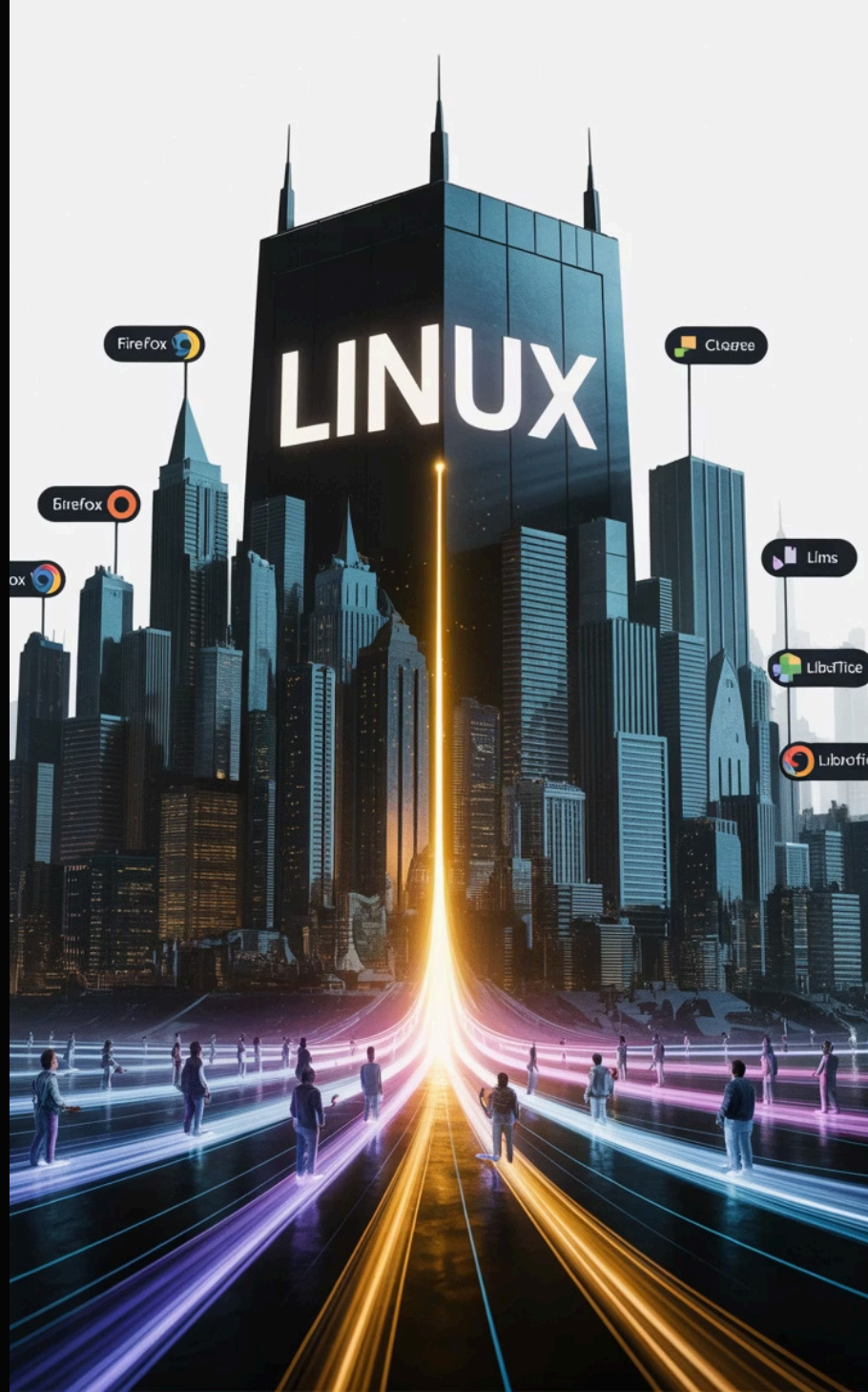
Safety Tips

- Always check for errors in your scripts
- Use the BEGIN/END probes for setup/cleanup
- Be aware of context - who is running the probe?
- Remember that kprobes can break between kernel versions
- Use duration limits or interval:s:N { exit(); } for auto-termination

Mapping Kernel to Userspace Functions

Understanding the connection between kernel and userspace activity is crucial for performance analysis.

```
# Trace a kernel function and correlate with userspace
bpftrace -e 'kprobe:do_sys_open {
 printf("PID %d (%s) called do_sys_open from:\n", pid, comm);
 print(ustack);
 }'

# Trace userspace function and see which kernel functions it calls
bpftrace -e 'uprobe:/lib/x86_64-linux-gnu/libc.so.6:fopen {
 @start[tid] = nsecs;
 }
 kprobe:do_sys_open /@start[tid]/ {
 printf("PID %d (%s) fopen -> do_sys_open\n", pid, comm);
 }
 uretprobe:/lib/x86_64-linux-gnu/libc.so.6:fopen /@start[tid]/ {
 delete(@start[tid]);
 }'
```



Advanced Kernel-Userspace Correlation

```
# Map fopen() to open() syscall and track latency
bpftrace -e '
 // Track when libc fopen() is called
 uprobe:/lib/x86_64-linux-gnu/libc.so.6:fopen {
 printf("fopen(%s) called from userspace\n", str(arg0));
 @fopen_args[tid] = str(arg0);
 @fopen_start[tid] = nsecs;
 }

 // Track when open syscall happens
 tracepoint:syscalls:sys_enter_open /@fopen_start[tid]/ {
 printf(" -> open(%s) syscall from fopen\n", str(args->filename));
 @open_start[tid] = nsecs;
 }

 // Track when open syscall completes
 tracepoint:syscalls:sys_exit_open /@open_start[tid]/ {
 printf(" <- open() returned %d after %d ns\n",
 args->ret, nsecs - @open_start[tid]);
 delete(@open_start[tid]);
 }

 // Track when fopen() returns
 uretprobe:/lib/x86_64-linux-gnu/libc.so.6:fopen /@fopen_start[tid]/ {
 printf("<- fopen(%s) returned after %d ns\n",
 @fopen_args[tid], nsecs - @fopen_start[tid]);
 delete(@fopen_args[tid]);
 delete(@fopen_start[tid]);
 }
'
```

Function-to-Syscall Mapper

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

// libc_syscall_mapper.bt - Map libc functions to syscalls
//
// This script traces commonly used libc functions and shows which
// syscalls they trigger, helping understand the relationship between
// userspace APIs and kernel operations.

BEGIN {
 printf("Tracing libc functions to syscall mapping...\n");
 printf("Hit Ctrl-C to end.\n\n");
}

// File I/O functions
uprobe:/lib/x86_64-linux-gnu/libc.so.6:fopen,
uprobe:/lib/x86_64-linux-gnu/libc.so.6:fread,
uprobe:/lib/x86_64-linux-gnu/libc.so.6:fwrite,
uprobe:/lib/x86_64-linux-gnu/libc.so.6:fclose {
 $func = strncmp(probe, "uprobe:/lib/x86_64-linux-gnu/libc.so.6:", 100);
 @func[tid] = $func;
 @start[tid] = nsecs;
 @call_count[$func] += 1;
}

// Track all syscalls while a libc function is executing
tracepoint:syscalls:sys_enter_* /@start[tid]/ {
 $syscall = substr(probe, sizeof("tracepoint:syscalls:sys_enter_") - 1);
 $elapsed = (nsecs - @start[tid]) / 1000; // microseconds

 // Map the libc function to this syscall
 @func_to_syscall[@func[tid], $syscall] += 1;

 // Record detailed call (for first few calls only)
 if (@detailed_calls < 20) {
 printf("%-6d %-16s %-6d %-20s -> %-20s\n",
 @detailed_calls, comm, pid, @func[tid], $syscall);
 @detailed_calls += 1;
 }
}

// Function returns
uretprobe:/lib/x86_64-linux-gnu/libc.so.6:fopen,
uretprobe:/lib/x86_64-linux-gnu/libc.so.6:fread,
uretprobe:/lib/x86_64-linux-gnu/libc.so.6:fwrite,
uretprobe:/lib/x86_64-linux-gnu/libc.so.6:fclose /@start[tid]/ {
 $func = strncmp(probe, "uretprobe:/lib/x86_64-linux-gnu/libc.so.6:", 100);
 $elapsed = (nsecs - @start[tid]) / 1000; // microseconds

 // Record latency
 @func_latency[$func] = hist($elapsed);
 delete(@func[tid]);
 delete(@start[tid]);
}

END {
 printf("\n=== Function to Syscall Mapping ===\n");
 printf("\nLibc function call counts:\n");
 print(@call_count);

 printf("\nLibc function to syscall mapping (which syscalls each libc function calls):\n");
 print(@func_to_syscall);

 printf("\nLibc function latency histograms (microseconds):\n");
 print(@func_latency);
}
```

Process Event Probes

bpftrace can monitor the entire lifecycle of processes from creation to exit:

```
# Process creation and exit
bpftrace -e 'tracepoint:sched:sched_process_fork {
 printf("Process created: parent %s (%d) -> child %s (%d)\n",
 args->parent_comm, args->parent_pid,
 args->child_comm, args->child_pid);
 }
 tracepoint:sched:sched_process_exit {
 printf("Process exit: %s (%d) exit code %d\n",
 args->comm, args->pid, args->exit_code);
 }'

# Track process lifecycle events (longer running)
bpftrace -e '
 tracepoint:sched:sched_process_fork { printf("%-8s %s (%d) -> %s (%d)\n", "FORK",
 args->parent_comm, args->parent_pid,
 args->child_comm, args->child_pid); }
 tracepoint:sched:sched_process_exec { printf("%-8s %s (%d) exec %s\n", "EXEC",
 args->comm, args->pid, args->filename); }
 tracepoint:sched:sched_process_exit { printf("%-8s %s (%d) exit_code=%d\n", "EXIT",
 args->comm, args->pid, args->exit_code); }
 tracepoint:sched:sched_switch { printf("%-8s %s (%d) -> %s (%d)\n", "SWITCH",
 args->prev_comm, args->prev_pid,
 args->next_comm, args->next_pid); }'
```

Process Creation Tracer

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

// processtree.bt - Trace process creation and build a process tree

BEGIN {
 printf("Tracing process creation... Hit Ctrl-C to end.\n");
 printf("%-20s %-7s %-7s %s\n", "TIME", "PPID", "PID", "COMMAND");
}

tracepoint:sched:sched_process_fork {
 $ppid = args->parent_pid;
 $pid = args->child_pid;
 $pcomm = args->parent_comm;
 $ccomm = args->child_comm;

 // Record relationship for tree building
 @parent[$pid] = $ppid;

 // Record creation time
 $ts = nsecs / 1000000000; // Convert to seconds
 time("%H:%M:%S ", $ts);
 printf("%-7d %-7d %s -> %s\n", $ppid, $pid, $pcomm, $ccomm);
}

tracepoint:sched:sched_process_exec {
 $pid = args->pid;
 $comm = args->comm;
 $filename = args->filename;

 // Record command line arguments
 $ts = nsecs / 1000000000;
 time("%H:%M:%S ", $ts);
 printf("%-7s %-7d %s execed %s",
 @parent[$pid] ? str(@parent[$pid]) : "?",
 $pid, $comm, $filename);

 // Add args if available
 $argp = (const char **)args->argv;
 $argstr = "";
 if ($argp) {
 $i = 1;
 $arg = (const char *)$argp[$i];
 while ($arg) {
 $argstr = strcat($argstr, " ");
 $argstr = strcat($argstr, str($arg));
 $i++;
 $arg = (const char *)$argp[$i];
 }
 }
 printf("%s\n", $argstr);
}

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

 $ts = nsecs / 1000000000;
 time("%H:%M:%S ", $ts);
 printf("%-7s %-7d %s exited with code %d\n",
 @parent[$pid] ? str(@parent[$pid]) : "?",
 $pid, $comm, $exit_code);
}

END {
 printf("\nProcess hierarchy (PPID -> PID counts):\n");
 print(@parent);
}
```

Debugging bpftrace Scripts

When developing bpftrace scripts, you can use the following debugging techniques:

Print Statements

Add printf() statements to see values during execution:

```
printf("PID: %d, Filename: %s\n", pid, @filename[tid]);
```

Dry Run Mode

Test compilation without running:

```
bpftrace -d script.bt
```

Variable Inspection

Print maps and variables periodically:

```
interval:s:1 { print(@mymap); }
```

Verbose Mode

Enable verbose output for troubleshooting:

```
bpftrace -v script.bt
```

Always remember to check variable types, scope, and clean up resources with delete() when no longer needed.

Workload Characterization

Understand workload patterns to optimize system performance:

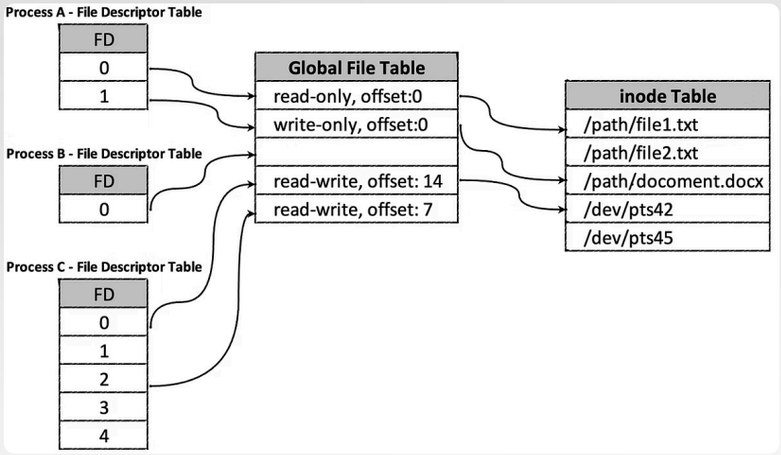
```
# CPU time by process
bpftrace -e 'profile:hz:99 {
    @cpu[comm] = count();
}'

# I/O pattern by process
bpftrace -e 'kprobe:vfs_read {
    @read_bytes[comm] = sum(arg2);
}
kprobe:vfs_write {
    @write_bytes[comm] = sum(arg2);
}'

# Memory allocation by process
bpftrace -e 'kretprobe:kmalloc {
    @bytes[comm] = sum(retval);
}'
```

Patterns to look for:

- CPU-bound vs I/O-bound workloads
- Memory access patterns (sequential vs random)
- System call frequency and types
- Resource contention between processes

Workload Profiler Script

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

// workload_profile.bt - Comprehensive workload profiling

BEGIN {
 printf("Starting workload profiling... Hit Ctrl-C to end.\n");
 @start_time = nsecs;
}

// CPU usage
profile:hz:99 {
 @cpu_samples[comm] += 1;

 // Stack sampling (limit to interesting processes)
 if (comm != "swapper" && comm != "bpftrace") {
 @stacks[comm, kstack, ustack] += 1;
 }
}

// Memory allocation tracking
kprobe:kmalloc,
kprobe:kmem_cache_alloc {
 @mem_alloc_bytes[comm] += arg1;
 @mem_alloc_count[comm] += 1;
}

kprobe:kfree,
kprobe:kmem_cache_free {
 @mem_free_count[comm] += 1;
}

// File I/O tracking
kprobe:vfs_read {
 @read_calls[comm] += 1;
 @read_bytes[comm] += arg2;
}

kprobe:vfs_write {
 @write_calls[comm] += 1;
 @write_bytes[comm] += arg2;
}

// Network I/O tracking
kprobe:sock_sendmsg {
 @net_tx_calls[comm] += 1;
}

kprobe:sock_recvmsg {
 @net_rx_calls[comm] += 1;
}

// System call tracking
tracepoint:syscalls:sys_enter_* {
 @syscalls[comm, probe] += 1;
}

// Periodic output
interval:s:10 {
 time("%H:%M:%S Workload profile snapshot:\n");

 printf("\nTop 10 CPU consumers:\n");
 print(@cpu_samples, 10);

 printf("\nTop 10 memory allocators (bytes):\n");
 print(@mem_alloc_bytes, 10);

 printf("\nTop 10 I/O processes (read bytes):\n");
 print(@read_bytes, 10);

 printf("\nTop 10 I/O processes (write bytes):\n");
 print(@write_bytes, 10);

 clear(@cpu_samples);
 clear(@mem_alloc_bytes);
 clear(@mem_alloc_count);
 clear(@mem_free_count);
 clear(@read_bytes);
 clear(@read_calls);
 clear(@write_bytes);
 clear(@write_calls);
 clear(@net_tx_calls);
 clear(@net_rx_calls);
}

END {
 $runtime = (nsecs - @start_time) / 1000000000;
 printf("\n=== Final Workload Profile (Runtime: %.2f seconds) ===\n", $runtime);

 printf("\nTop syscalls by process:\n");
 print(@syscalls, 20);

 printf("\nTop stack traces:\n");
 print(@stacks, 10);
}
```



Real-World Example: Finding File Descriptor Leaks

File descriptor leaks can cause "too many open files" errors. Let's detect them:

```
# Track processes that open files but don't close them
bpftrace -e '
 tracepoint:syscalls:sys_enter_open {
 @opens[pid, comm] += 1;
 }
 tracepoint:syscalls:sys_enter_close {
 @closes[pid, comm] += 1;
 }
 interval:s:5 {
 printf("Potential FD leaks (opens - closes > 10):\n");
 foreach ($p in @opens) {
 $pid = $p[0];
 $comm = $p[1];
 $open_count = @opens[$pid, $comm];
 $close_count = @closes[$pid, $comm];
 if ($open_count - ($close_count || 0) > 10) {
 printf(" %s (PID %d): opens: %d, closes: %d, diff: %d\n",
 $comm, $pid, $open_count, $close_count || 0,
 $open_count - ($close_count || 0));
 }
 }
 }'
```

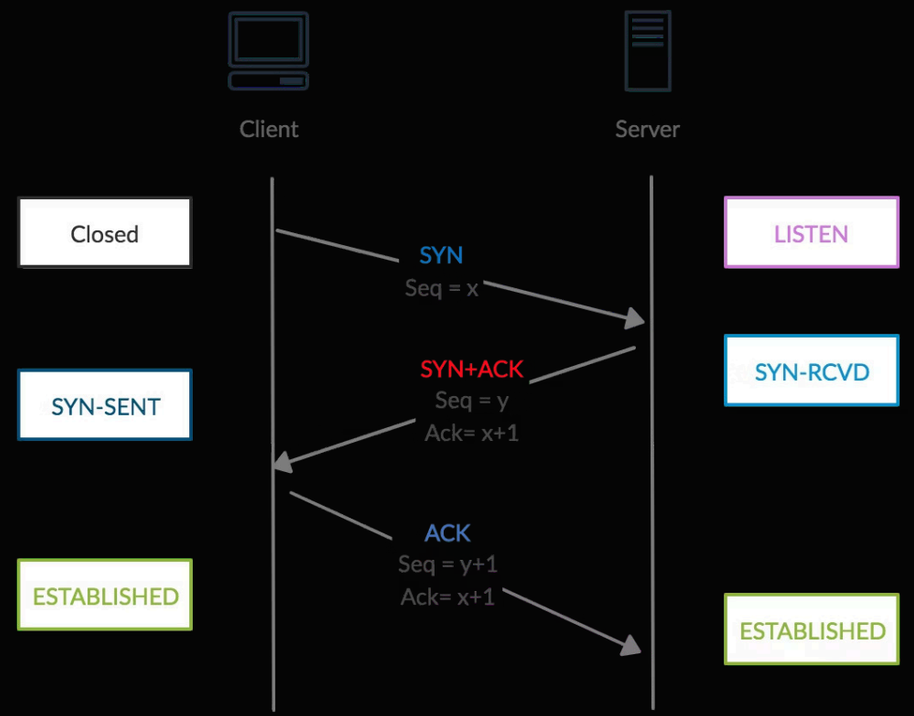
File Descriptor Leak Detector

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

// fd_leak_detector.bt - Detect potential file descriptor leaks

BEGIN {
 printf("Monitoring for file descriptor leaks...\n");
 printf("Hit Ctrl-C to end and show summary.\n\n");
}

// Track file open operations
tracepoint:syscalls:sys_enter_open,
tracepoint:syscalls:sys_enter_openat {
 @opens[pid, comm] += 1;

 // Store filename for recently opened files
 if (@recent_files_count < 100) {
 @recent_files[pid, comm, nsecs] = str(args->filename);
 @recent_files_count += 1;
 }
}

// Track file close operations
tracepoint:syscalls:sys_enter_close {
 @closes[pid, comm] += 1;
}

// Track process exit to account for FD cleanup on exit
tracepoint:sched:sched_process_exit {
 // When a process exits, all its FDs are automatically closed
 // We can remove it from our tracking
 @exited[args->pid] = 1;
}

// Periodically check for potential leaks
interval:s:10 {
 printf("\n=== Potential FD leak check at ");
 time("%H:%M:%S ===\n");

 // For each process, calculate the difference between opens and closes
 foreach ($p in @opens) {
 $pid = $p[0];
 $comm = $p[1];

 // Skip processes that have exited
 if (@exited[$pid]) {
 continue;
 }

 $open_count = @opens[$pid, $comm];
 $close_count = @closes[$pid, $comm] || 0;
 $diff = $open_count - $close_count;

 // Flag potential leaks (threshold can be adjusted)
 if ($diff > 20) {
 printf("WARNING: Potential FD leak in %s (PID %d)\n", $comm, $pid);
 printf(" Opens: %d, Closes: %d, Diff: %d\n",
 $open_count, $close_count, $diff);

 // Show some recently opened files by this process
 printf(" Recently opened files:\n");
 foreach ($f in @recent_files) {
 $file_pid = $f[0];
 $file_comm = $f[1];

 if ($file_pid == $pid && $file_comm == $comm) {
 printf(" %s\n", @recent_files[$f]);
 }
 }
 printf("\n");
 }
 }

 // Limit the size of the recent files map
 @recent_files_count = 0;
 clear(@recent_files);
}

END {
 printf("\n=== FD Leak Detection Summary ===\n");
 printf("\nProcesses with largest open-close differential:\n");

 // Sort processes by the difference between opens and closes
 foreach ($p in @opens) {
 $pid = $p[0];
 $comm = $p[1];

 // Skip processes that have exited
 if (@exited[$pid]) {
 continue;
 }

 $open_count = @opens[$pid, $comm];
 $close_count = @closes[$pid, $comm] || 0;
 $diff = $open_count - $close_count;

 @fd_diff[$comm, $pid] = $diff;
 }

 print(@fd_diff, 20);
}
```

Real-World Example: Network Latency Analysis

Network latency issues can be difficult to diagnose. Let's trace TCP connections:

```
# Trace TCP connect latency
bpftrace -e '
    kprobe:tcp_v4_connect,
    kprobe:tcp_v6_connect {
        @start[tid] = nsecs;
    }

    kretprobe:tcp_v4_connect,
    kretprobe:tcp_v6_connect /@start[tid]/ {
        $latency = nsecs - @start[tid];
        @connect_latency = hist($latency / 1000000); // ms
        delete(@start[tid]);
    }'

# Trace TCP connection setup time
bpftrace -e '
    kprobe:tcp_connect {
        @start[arg0] = nsecs;
    }

    kprobe:tcp_finish_connect /@start[arg0]/ {
        @conn_setup = hist((nsecs - @start[arg0]) / 1000000);
        delete(@start[arg0]);
    }'
```



Network protocol stacks are complex, with many potential bottlenecks:

- Socket creation overhead
- DNS resolution time
- TCP handshake latency
- Packet transmission time
- Application processing delay

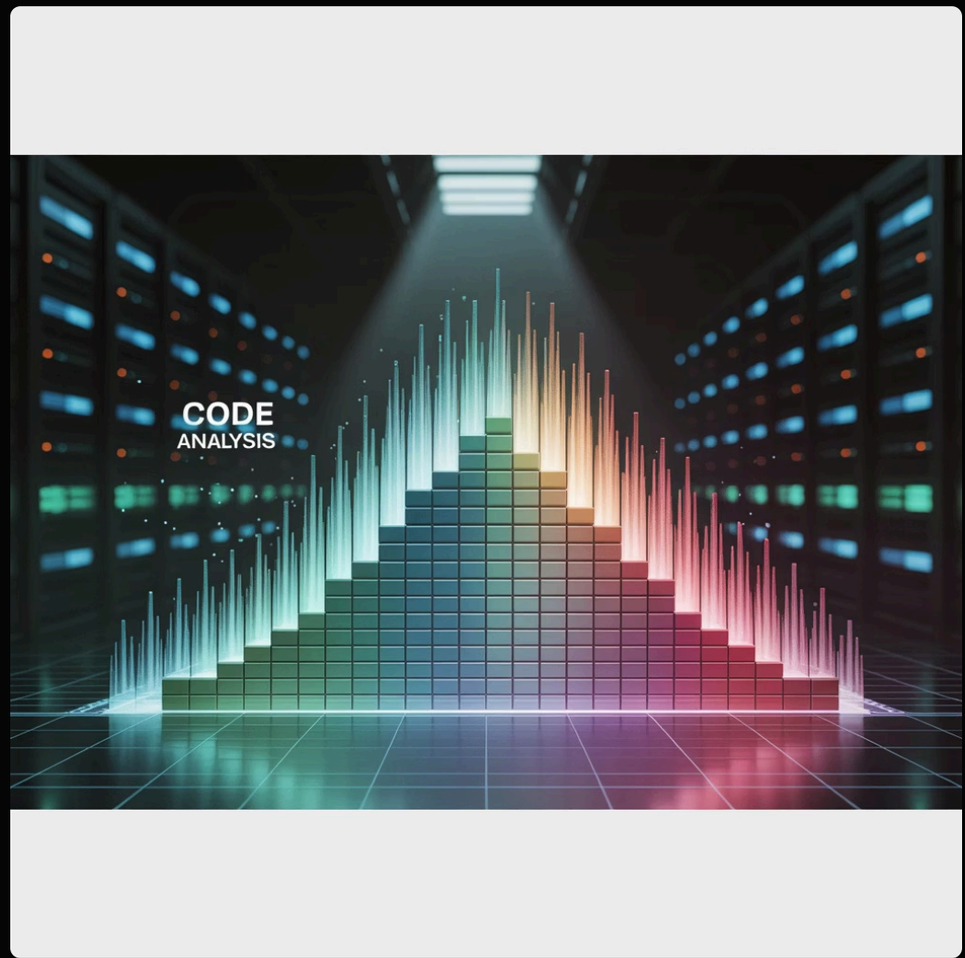
Network Latency Analyzer

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

// tcp_latency.bt - Analyze TCP connection and transmission latency

BEGIN {
 printf("Tracing TCP latency... Hit Ctrl-C to end.\n");
}

// TCP connection attempt
kprobe:tcp_v4_connect,
kprobe:tcp_v6_connect {
 @conn_start[tid] = nsecs;
}

// TCP connection completion
kretprobe:tcp_v4_connect,
kretprobe:tcp_v6_connect /@conn_start[tid]/ {
 $duration = nsecs - @conn_start[tid];
 @tcp_connect_latency_ms = hist($duration / 1000000);
 delete(@conn_start[tid]);
}

// TCP packet send
kprobe:tcp_sendmsg {
 // Track socket and size
 @send_start[tid] = nsecs;
 @send_size[tid] = arg2;
}

kretprobe:tcp_sendmsg /@send_start[tid]/ {
 $duration = nsecs - @send_start[tid];
 $size = @send_size[tid];

 // Only count successful sends
 if (retval >= 0) {
 @tcp_send_latency_us[$size < 1024 ? "small (<1KB)" :
 $size < 10240 ? "medium (1-10KB)" :
 "large (>10KB)"] = hist($duration / 1000);
 }

 delete(@send_start[tid]);
 delete(@send_size[tid]);
}

// TCP packet receive
kprobe:tcp_recvmsg {
 @recv_start[tid] = nsecs;
}

kretprobe:tcp_recvmsg /@recv_start[tid]/ {
 $duration = nsecs - @recv_start[tid];

 // Only count successful receives
 if (retval > 0) {
 @tcp_recv_latency_us[retval < 1024 ? "small (<1KB)" :
 retval < 10240 ? "medium (1-10KB)" :
 "large (>10KB)"] = hist($duration / 1000);
 }

 delete(@recv_start[tid]);
}

// Track TCP retransmits
kprobe:tcp_retransmit_skb {
 @retransmits[comm] += 1;
}

// Track application processes and their TCP operations
kprobe:tcp_sendmsg,
kprobe:tcp_recvmsg {
 @tcp_by_process[comm, probe] += 1;
}

interval:s:10 {
 printf("\n=== TCP Retransmit Count (potential network issues) ===\n");
 print(@retransmits);
 clear(@retransmits);

 printf("\n=== TCP Operations by Process ===\n");
 print(@tcp_by_process, 10);
 clear(@tcp_by_process);
}

END {
 printf("\n=== TCP Connection Latency (milliseconds) ===\n");
 print(@tcp_connect_latency_ms);

 printf("\n=== TCP Send Latency by Size (microseconds) ===\n");
 print(@tcp_send_latency_us);

 printf("\n=== TCP Receive Latency by Size (microseconds) ===\n");
 print(@tcp_recv_latency_us);
}
```

Debugging High CPU Usage

When a system experiences high CPU usage, bpftrace can help identify the cause:

```
# Sample running processes on all CPUs
bpftrace -e 'profile:hz:99 {
        @[cpu, comm] = count();
    }'

# Sample kernel stacks to find hotspots
bpftrace -e 'profile:hz:99 {
        @[kstack] = count();
    }'

# See which processes are using most CPU with stack traces
bpftrace -e 'profile:hz:99 {
        @[comm, kstack, ustack] = count();
    }'

# Count time spent in different scheduler states
bpftrace -e 'tracepoint:sched:sched_switch {
        @[args->prev_state] = count();
    }'
```



Key metrics to look for:

- User vs. kernel CPU time ratio
- System call overhead
- Interrupt processing time
- Context switch frequency
- CPU run queue length

CPU Hotspot Analyzer

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

// cpu_hotspot.bt - Identify CPU usage hotspots

BEGIN {
 printf("Analyzing CPU hotspots... Hit Ctrl-C after 30+ seconds of data collection.\n");
 printf("Sampling across all CPUs at 99Hz...\n");
 @start_time = nsecs;
}

// Sample CPUs at 99Hz to minimize overhead while getting good data
profile:hz:99 {
 // Track process name
 @process_samples[comm] += 1;

 // Track per-CPU usage
 @cpu_samples[cpu] += 1;

 // Track kernel vs. user time
 @mode[comm, "kernel"] += kstack != 0;
 @mode[comm, "user"] += ustack != 0;

 // Collect stacks, but only for processes over a threshold
 // to avoid collecting too much data
 if (@process_samples[comm] > 10) {
 if (kstack != 0) {
 @kstacks[comm, kstack] += 1;
 }
 if (ustack != 0) {
 @ustacks[comm, ustack] += 1;
 }
 }
}

// Track scheduler events
tracepoint:sched:sched_switch {
 // Track context switches
 @context_switches += 1;

 // Track scheduler states
 @sched_state[args->prev_state] += 1;

 // Track run queue contention
 @task_switches[args->prev_comm, args->next_comm] += 1;
}

// Track CPU migrations
tracepoint:sched:sched_migrate_task {
 @cpu_migrations[args->comm, args->orig_cpu, args->dest_cpu] += 1;
}

interval:s:5 {
 printf("\n=== CPU Usage by Process (top 10) ===\n");
 print(@process_samples, 10);

 printf("\n=== CPU Usage by Core ===\n");
 print(@cpu_samples);

 printf("\n=== Context Switch Count: %d ===\n", @context_switches);

 clear(@process_samples);
 clear(@cpu_samples);
 clear(@context_switches);
}

END {
 $runtime = (nsecs - @start_time) / 1000000000;
 printf("\n=== CPU Hotspot Analysis (Runtime: %.2f seconds) ===\n", $runtime);

 printf("\n=== Kernel vs. User Time by Process ===\n");
 print(@mode);

 printf("\n=== Scheduler States ===\n");
 print(@sched_state);

 printf("\n=== CPU Migrations ===\n");
 print(@cpu_migrations, 10);

 printf("\n=== Common Task Switches (Potential Contention) ===\n");
 print(@task_switches, 10);

 printf("\n=== Top Kernel Stacks (Kernel Hotspots) ===\n");
 print(@kstacks, 10);

 printf("\n=== Top User Stacks (Application Hotspots) ===\n");
 print(@ustacks, 10);

 printf("\nTo further analyze a specific process, try:\n");
 printf("bpftrace -e 'profile:hz:99 /comm == \"PROCESS_NAME\"/ { @[ustack] = count(); }'\n");
}
```

Performance Tuning Opportunities

Observe

Use bpftrace to identify bottlenecks:

- System call latency
- I/O patterns
- CPU hotspots
- Memory allocation

Target

Identify specific areas for improvement:

- File descriptors leaks
- Excessive context switches
- Memory fragmentation
- Inefficient I/O patterns

Tune

Implement changes with measurable goals:

- Kernel parameters
- Application changes
- Resource limits
- Workload distribution

Verify

Measure the impact of changes:

- Before/after latency comparison
- Resource utilization improvements
- Error rate reduction
- Overall system throughput

The key to effective tuning is measuring the right metrics before and after changes to quantify improvements.

Key Takeaways

- [ ] bpftrace is a powerful observability tool

  With minimal overhead, bpftrace provides visibility into virtually every aspect of Linux system behavior.

- [ ] One-liners enable quick investigations

  For rapid troubleshooting, one-liners provide immediate insight into system behavior without complex setup.

- [ ] Comprehensive scripts for deeper analysis

  For thorough performance analysis, scripts can correlate events across subsystems and provide rich data visualization.

- [ ] Dynamic observability without reboots

  Unlike traditional tools, bpftrace enables observing production systems without disruption or configuration changes.

- [ ] Next steps: Advanced tracing techniques

  Continue to Module 3 to learn about advanced tracing of network subsystems, memory management, and application profiling.