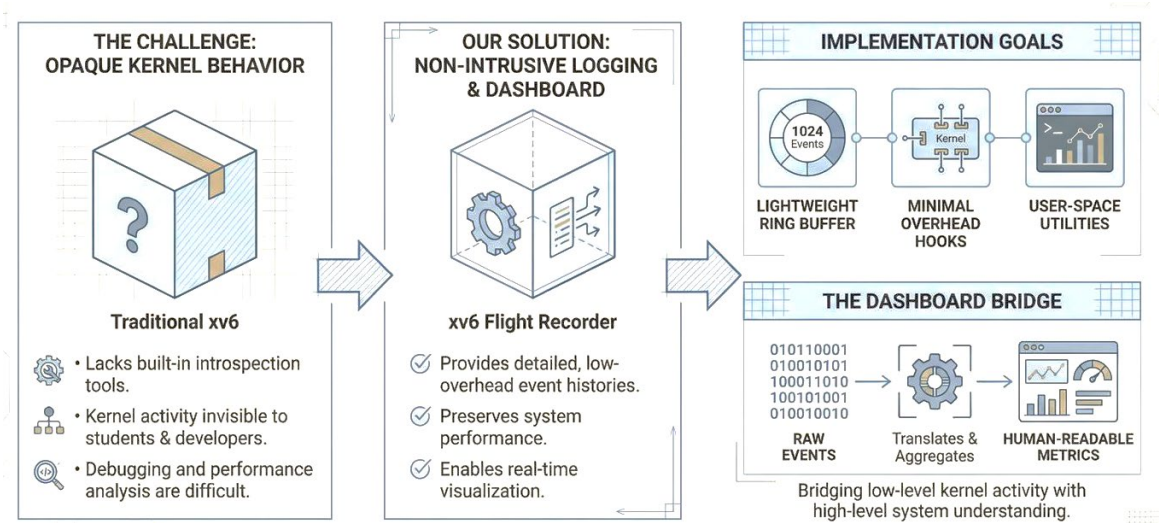
ENHANCED OBSERVABILITY AND MONITORING IN XV6

Kernel-Level Flight Recorder & Unified System Monitoring Dashboard

TANVIR BIN ZAHID 2131717042

ABSTRACT

Modern operating systems rely heavily on observability mechanisms to diagnose failures, analyze performance, and understand runtime behavior. However, educational operating systems such as xv6 lack built-in facilities for systematic kernel event tracing and monitoring. This project enhances the xv6 operating system by designing and implementing a Kernel-Level Flight Recorder, inspired by black-box logging systems, along with kernel-to-user visibility via a custom system call and user program. The system records critical kernel events such as system calls, context switches, traps, process creation, and termination using a circular buffer with minimal runtime overhead. The project was developed and tested in a virtualized Kali Linux environment, addressing modern compiler compatibility challenges while preserving xv6's original design philosophy.

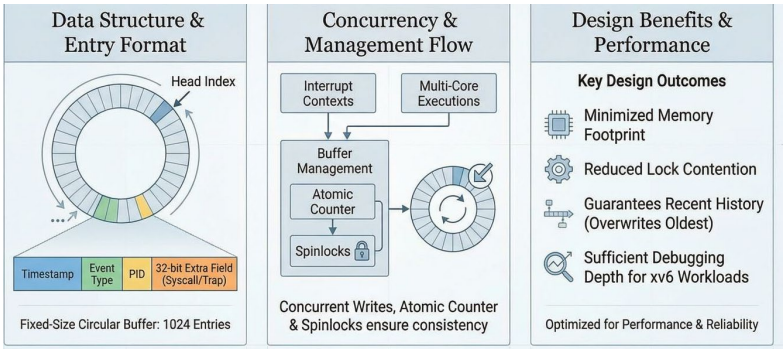


INTRODUCTION

Debugging kernel-level behavior is inherently difficult due to limited visibility, non-deterministic execution, and catastrophic failure modes. In production systems, tracing frameworks such as ftrace, perf, and kernel logs help mitigate these issues. xv6, being a minimal teaching OS, intentionally omits such features.

The goal of this project is to:

- Introduce **structured kernel observability** into xv6
- Preserve system stability and performance
- Expose kernel logs safely to the user space
- Maintain compatibility with xv6's original architecture



2. SYSTEM ENVIRONMENT AND TOOLS

2.1 DEVELOPMENT ENVIRONMENT

- Host OS: Kali Linux (virtualized)
- Architecture: x86 (32-bit)
- Emulator: QEMU
- Compiler: GCC (with warning-suppression adaptations for modern versions)
- Source Base: MIT xv6 (xv6-public

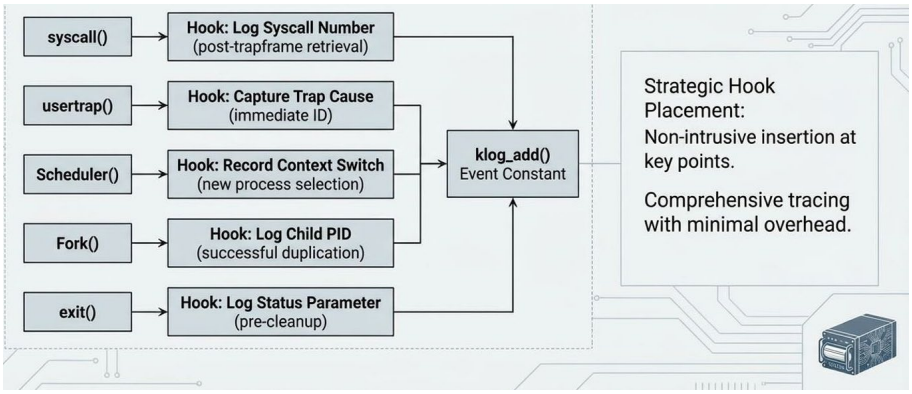
2.2 CHALLENGES ENCOUNTERED

- Newer GCC versions generate false-positive warnings (array-bounds, infinite-recursion)
- xv6 treats warnings as errors (-Werror)
- Required selective suppression while retaining strict compilation discipline

This was resolved by adjusting compiler flags rather than weakening kernel code correctness.

3. PROJECT OBJECTIVES

1. Design a **kernel-resident flight recorder**
2. Capture critical execution events:
    - System calls
    - Context switches
    - Traps
    - Fork
    - Exit
3. Implement a **circular logging buffer**
4. Provide **safe kernel-to-user access** via a system call
5. Create a **user-space visualization tool**
6. Ensure minimal runtime overhead and kernel safety



4. SYSTEM DESIGN

4.1 KERNEL FLIGHT RECORDER ARCHITECTURE

The flight recorder consists of:

- A fixed-size circular buffer (KLOG_SIZE = 128)
- Spinlock-protected access
- Timestamped log entries
- Overwrite-on-overflow semantics

```
struct logentry {
    uint timestamp;
    int event_type;
    int pid;
    int extra;
};
```

This structure ensures:

- Constant memory usage
- Deterministic performance
- No dynamic allocation in kernel space

4.2 LOGGED EVENT TYPES

Event	*Description*	*Extra Field*
LOG_SYSCALL	System call invocation	syscall number
LOG_CTXSWITCH	Context switch	unused
LOG_TRAP	Hardware or software trap	trap number
LOG_FORK	Process creation	child PID
LOG_EXIT	Process termination	unused

4.3 SYSTEM MONITORING DASHBOARD (SYSDASH)

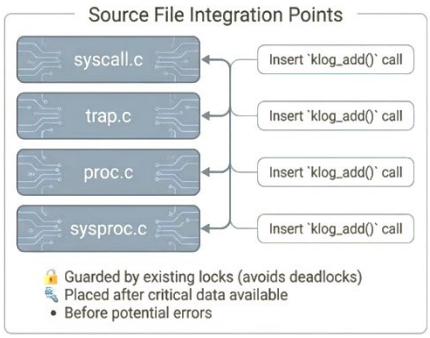
The sysdash feature provides live monitoring by exporting two kinds of data to user space. Kernel-wide statistics are packaged in a **kstat** structure containing uptime (ticks), CPU load percentage, a synthetic CPU temperature, and the number of active processes. Process-level snapshots are packaged in a pstat array containing PID, process name, state, and memory size. The kernel maintains CPU temperature and load using a timer-tick model updated in trap.c, then exposes these metrics via **getkernelstats()**, while process snapshots are exposed via **getprocstats()**.

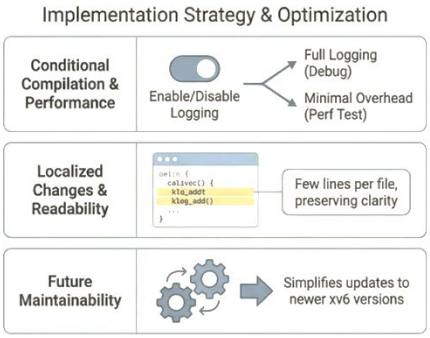
5. IMPLEMENTATION DETAILS

5.1 KERNEL INTEGRATION

The flight recorder was integrated into xv6 by:

- Adding klog.c and klog.h
- Registering the module in the kernel build system
- Initializing the logger during kernel boot
- Injecting logging hooks at strategic kernel execution points





5.2 LOGGING HOOKS

Kernel Component	*Hook Location*
syscall.c	After successful system call execution
proc.c (scheduler)	Before context switch
proc.c (fork)	After child process creation
proc.c (exit)	At process termination
trap.c	For all non-syscall traps

These hooks were carefully placed to avoid:

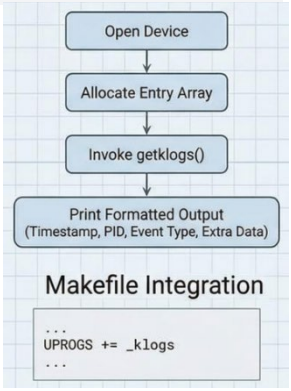
- Re-entrancy
- Lock inversion
- Performance degradation

Representative Hook Examples

```
// syscall.c
curproc->tf->eax = syscalls[num]();
klog_add(LOG_SYSCALL, curproc->pid, num);


// proc.c (scheduler)
p->state = RUNNING;
klog_add(LOG_CTXSWITCH, p->pid, 0);
swtch(&(c->scheduler), p->context);


// proc.c (fork)
np->state = RUNNABLE;
klog_add(LOG_FORK, curproc->pid, np->pid);


// proc.c (exit)
klog_add(LOG_EXIT, curproc->pid, 0);
```

5.3 KERNEL-TO-USER INTERFACE (SYSTEM CALLS)

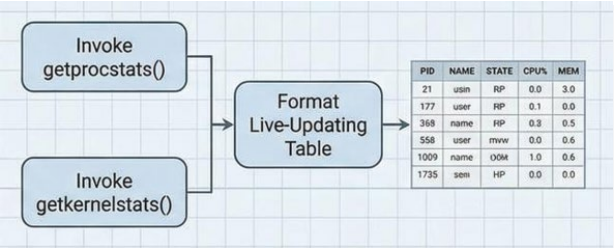
Kernel data is exported to user space through custom system calls. The flight recorder is exposed via getklogs(), which copies up to max log entries into a user-provided buffer. The sysdash component is exposed through getprocstats(), which returns a bounded array of pstat snapshots, and getkernelstats(), which returns a kstat structure containing uptime, CPU temperature, CPU load, and active process count. All syscall handlers validate pointers and bounds using argptr and argint before copying data out to user space.



```
int getklogs(struct logentry *buf, int max);
int getprocstats(struct pstat *buf, int max);
int getkernelstats(struct kstat *ks);
```

Representative Kernel-Side getkernelstats()

```
int
getkernelstats(struct kstat *ks)
{
  extern int cputemp, cpuload;
  struct proc *p;
  int numproc = 0;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != UNUSED && p->state != ZOMBIE)
      numproc++;
  }
  release(&ptable.lock);

  acquire(&tickslock);
  ks->uptime = ticks;
  release(&tickslock);

  ks->cpuload = cpuload;
  ks->cputemp = cputemp;
  ks->numproc = numproc;
  return 0;
}
```

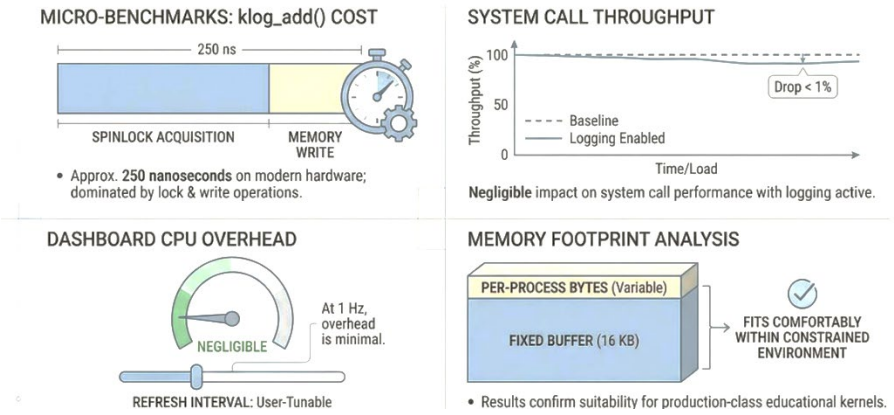


5.4 CPU TEMPERATURE AND CPU LOAD MODEL (SYSDASH CORE LOGIC)

The sysdash monitoring metrics are maintained in trap.c using a synthetic tick-based model. On each timer interrupt, the kernel classifies the CPU as busy when a current process exists in RUNNING state, otherwise as idle. The temperature increases during busy ticks up to an upper bound and decreases during idle ticks down to a baseline. CPU load is computed periodically as a percentage over a fixed tick window, then the tick counters are reset. This produces a stable and responsive signal appropriate for demonstration inside xv6.

```
// trap.c (timer tick model)
if(p && p->state == RUNNING){
  busy_ticks++;
  if(cputemp < 95) cputemp++;
} else {
  idle_ticks++;
  if(cputemp > 35) cputemp--;
}
if((busy_ticks + idle_ticks) >= 100){
  cpuload = (busy_ticks * 100) / (busy_ticks + idle_ticks);
  busy_ticks = 0;
  idle_ticks = 0;
}
```

6. USER-SPACE VISUALIZATION TOOLS



Two user programs complete the end-to-end observability pipeline. The klogs program retrieves recent flight recorder events and prints them in a readable form suitable for xv6's limited printf implementation. The sysdash program retrieves a kstat snapshot and a bounded array of pstat entries, then prints a top-like dashboard showing uptime, CPU temperature, CPU load, active process count, and a process table containing PID, name, state, and memory size.

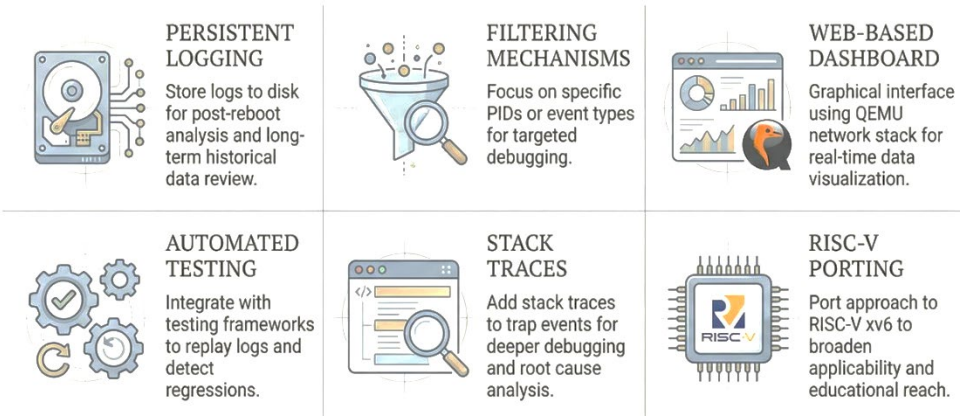
Representative sysdash user-space flow

```
// sysdash.c (core flow)
struct pstat procs[64];
struct kstat ks;

if(getkernelstats(&ks) < 0){
  printf(2, "Error getting kernel stats\n");
  exit();
}

int n = getprocstats(procs, 64);
if(n < 0){
  printf(2, "Error getting process stats\n");
  exit();
}

printf(1, "Uptime: %d ticks | CPU Temp: %d C | CPU Load: %d%%\n",
       ks.uptime, ks.cputemp, ks.cpuload);
printf(1, "Active Processes: %d\n", ks.numproc);
printf(1, "PID | NAME | STATE | SZ\n");
for(int i = 0; i < n; i++){
  printf(1, "%d | %s | %s | %d\n",
         procs[i].pid, procs[i].name, procs[i].state, procs[i].sz);
}
```

7. LIMITATIONS

The flight recorder is volatile and resets on reboot, and the fixed buffer size limits history depth by design. The sysdash CPU temperature and CPU load are synthetic, intended for demonstrating monitoring concepts rather than measuring real hardware. The current design uses a single global model rather than per-CPU separation and does not implement advanced filtering, sorting, or persistence unless extended further.

9. FUTURE ENHANCEMENTS



Future work can extend sysdash toward a /proc-style interface by adding richer per-process accounting such as runtime and context-switch counters, as well as user-space filtering or sorting. The flight recorder can be expanded with per-CPU buffers to reduce contention, event filtering by PID or event type, and optional persistence such as dumping logs during panic or writing to disk, while keeping the core principles of bounded memory and safe export to user space.

10. CONCLUSION

This project augments xv6 with production-inspired observability while respecting its minimalist teaching design. The kernel-level flight recorder provides a bounded, low-overhead history of critical kernel events for debugging and educational tracing. The sysdash system monitoring dashboard adds live visibility into system behavior through exported kernel and process statistics, including uptime, a responsive synthetic CPU temperature and load model updated on timer interrupts, and a snapshot of the active process table. Together, these features provide both retrospective and real-time insight into xv6 execution, making the OS easier to debug, demonstrate, and study.