Time-Lapse Image Acquisition & Frame Synchronization System

Javier Ferrer Ortega

University of Colorado Boulder - Coursera

Abstract

This report presents the design, analysis, and implementation of a hard real-time video processing system utilizing Rate Monotonic (RM) scheduling for deterministic frame acquisition, edge enhancement, and storage. The system employs three periodic services operating at different frequencies (10 Hz, 2 Hz, and 1 Hz) with fixed-priority preemptive scheduling to ensure predictable timing behavior. Through comprehensive schedulability analysis using the Liu & Layland utilization bound test and exact response time analysis, we demonstrate that the system meets all timing constraints with a total CPU utilization of 49%, providing substantial safety margins for all real-time tasks. The implementation leverages POSIX real-time extensions on Linux, including SCHED_FIFO scheduling, binary semaphores for inter-process communication, and dedicated CPU core assignment to achieve deterministic performance.

Keywords: Real-Time Systems, Rate Monotonic Analysis, Video Processing, Schedulability Analysis, POSIX Real-Time, Embedded Linux

Time-Lapse Image Acquisition & Frame Synchronization System

Real-time embedded systems represent a critical class of computing systems where correctness depends not only on the logical accuracy of computations but also on the time at which results are produced [1]. Unlike general-purpose computing systems that optimize for average-case performance or throughput, real-time systems must guarantee that specified timing constraints (deadlines) are met under all conditions. These systems are pervasive in modern society, appearing in applications ranging from automotive safety systems and industrial automation to medical devices and aerospace control systems. Real-time systems are classified into three categories based on the consequences of missing deadlines [2]: 1. Hard Real-Time Systems: Missing a deadline constitutes a system failure with potentially catastrophic consequences. Examples include airbag deployment systems, pacemakers, and anti-lock braking systems (ABS). 2. Firm Real-Time Systems: Missing occasional deadlines degrades system quality but does not cause catastrophic failure. However, late results have no utility. Examples include video streaming and telecommunications. 3. Soft Real-Time Systems: Deadlines are important for quality of service, but occasional violations are acceptable. Examples include multimedia playback and user interface responsiveness. Rate Monotonic (RM) scheduling, introduced by Liu and Layland in their seminal 1973 paper [3], represents one of the most widely used and theoretically sound approaches to real-time scheduling. RM is a fixed-priority preemptive scheduling algorithm where task priorities are assigned inversely proportional to their periods: tasks with shorter periods receive higher priorities. This priority assignment rule is provably optimal among all fixed-priority scheduling algorithms for periodic task sets with implicit deadlines (deadline equals period) [4]. The theoretical foundation of RM scheduling rests on several key principles: Optimality, Utilization Bound and Response Time Analysis (RTA): RTA computes the worst-case response time for each task by considering interference from all higher-priority tasks [5]. Video Processing as a Real-Time Application presents unique challenges for real-time embedded computing.

Modern cameras capture frames at fixed rates (e.g., 30 fps, 60 fps), creating periodic tasks with strict timing requirements. Frame acquisition must occur synchronously with the camera's frame rate to avoid frame loss, while subsequent processing and storage operations must complete before the next frame arrives or before their respective deadlines. The temporal characteristics of video processing workflows naturally align with the periodic task model assumed by RM scheduling theory. Frame acquisition operates at the camera's native frame rate, processing tasks operate at rates determined by computational requirements and application needs, and storage tasks operate at rates constrained by I/O subsystem performance. This hierarchical structure creates a multi-rate system ideally suited for RM analysis.

System Overview and Objectives

This report presents a real-time video processing system designed to capture, enhance, and store video frames with deterministic timing guarantees. The system architecture comprises three periodic services:- **Service 1 (Frame Acquisition)**: Operates at 10 Hz to capture frames from a USB camera using the Video4Linux2 (V4L2) interface- **Service 2 (Frame Processing)**: Operates at 2 Hz to perform Canny edge detection for feature enhancement- **Service 3 (Frame Storage)**: Operates at 1 Hz to write processed frames to persistent storage in PGM format. The primary objectives of this system are: 1. Deterministic Timing: Guarantee that all services meet their timing deadlines under worst-case conditions. 2. Zero Frame Loss: Ensure no frames are dropped during the 1801-frame capture sequence. 3. Quality Processing: Apply sophisticated edge detection algorithms without compromising real-time constraints. 4. Formal Verification: Provide mathematical proof of schedulability through RM analysis.

The system is implemented on embedded Linux using POSIX real-time extensions, including SCHED_FIFO scheduling policy, high-resolution timers, binary semaphores, and CPU affinity controls. A 100 Hz timer-driven sequencer provides precise temporal control over service activation, ensuring that each service is released at its designated frequency with minimal jitter.

Functional Requirements

The functional requirements define what the system must accomplish, independent of timing constraints. These requirements specify the core capabilities and behaviors expected from the video processing system.

1. Video Frame Acquisition

The system shall acquire video frames from a USB camera device (/dev/video0) using the Video4Linux2 (V4L2) API:

- Support YUYV pixel format (YUV 4:2:2 packed)
- Capture resolution: 640×480 pixels
- Extract luminance (Y) channel for grayscale processing
- Support memory-mapped I/O (mmap) for zero-copy frame access
- Implement burst reading to select the middle frame from buffered captures, reducing motion blur and temporal jitter

2. Edge Enhancement Processing

The system shall provide optional Canny edge detection for feature enhancement of acquired frames.

- Implement multi-stage Canny edge detection algorithm:
  - Gaussian blur (3×3 kernel) for noise reduction
  - Sobel gradient operators for edge detection
  - Dual-threshold edge classification (low: 30, high: 90)
- Support runtime enablement via command-line argument ("enhancement")
- Perform in-place processing to minimize memory overhead
- Alternative mode: frame differencing for motion detection

3. Persistent Frame Storage

The system shall store processed frames to the file system in Portable Gray Map (PGM) format:

- File naming: sequential numbering (frame_0001.pgm, frame_0002.pgm, ...)
- PGM format: P5 (binary), 8-bit grayscale, 640×480 resolution
- Storage location: configurable directory path (default: /home/.../frames)
- Atomic write operations: open → write header → write data → close
- Error handling: log failures but continue operation

4. System Initialization and Configuration

The system shall initialize all resources and verify configuration before beginning frame capture:

- Allocate ring buffer for frame storage (2048 frames × 307,200 bytes)
- Allocate auxiliary buffers (gray_prev, scratch_pad)
- Initialize V4L2 device with requested format and buffer count (10 buffers)
- Create and configure binary semaphores for inter-service synchronization
- Configure POSIX timer for 100 Hz sequencer
- Set real-time scheduling parameters (SCHED_FIFO, priorities, CPU affinity)

* Pre-allocation avoids dynamic memory allocation during time-critical operations, which could cause unpredictable delays.

5. Warm-Up and Calibration

The system shall implement a warm-up period before recording begins:

- Warm-up duration: 30 frames
- During warm-up: acquire frames but do not record to ring buffer
- Purpose: stabilize camera auto-exposure, auto-white-balance, and auto-focus
- Initialize baseline for frame differencing (if edge detection disabled)

6. Logging and Diagnostics

The system shall log timing information and operational events for analysis and debugging:

- Use syslog facility with LOG_CRIT priority for timing-critical events
- Log frame acquisition timestamps (relative to start time)
- Log processing events when edge detection is performed
- Log storage events when frames are written
- Include frame numbers and high-resolution timestamps (microsecond precision)

7. Graceful Termination

The system shall terminate cleanly after capturing the target number of frames.

- Target frame count: 1801 frames
- Signal all services to terminate when target reached
- Wait for all services to complete current operations (pthread_join)
- Release all allocated resources (memory, file descriptors, semaphores)
- Report final statistics (frames acquired, processed, stored)

Real-Time Requirements

Real-time requirements specify the temporal constraints that the system must satisfy to be considered correct. These constraints define deadlines, frequencies, and timing relationships between services.

Service Frequencies: The system shall execute services at specified frequencies with bounded jitter.

Worst-Case Execution Times (WCET): Each service shall complete execution within specified worst-case bounds.

Deadlines: All services shall use implicit deadlines (deadline equals period).

Priority Assignment: Service priorities shall be assigned according to Rate Monotonic principles (shorter period → higher priority).

Schedulability: The task set shall be provably schedulable under Rate Monotonic scheduling.

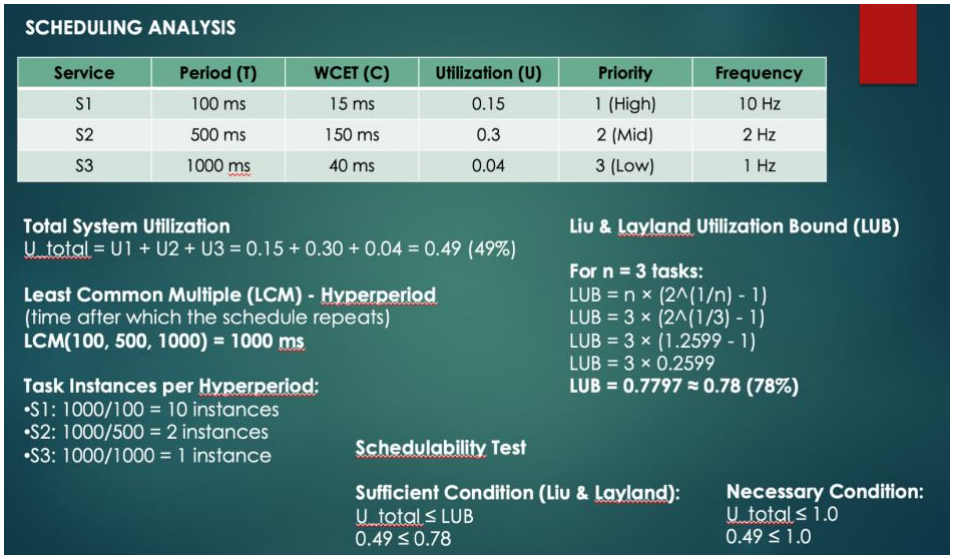
Preemption and Synchronization: The system shall support preemptive scheduling with bounded blocking times.

Timing Accuracy: The sequencer shall provide precise timing control with minimal drift.

- Sequencer frequency: 100 Hz (10 ms period)
- Timer source: CLOCK_REALTIME with POSIX timer_create/timer_settime
- Signal delivery: SIGALRM to sequencer handler
- Maximum drift: < 1 ms per 100 seconds
- Synchronization: services released via sem_post within ISR

Frame Loss Prevention: The system shall prevent frame loss due to buffer overruns or missed deadlines.

Figure 1. Real time requirements



Functional Design Overview and Diagrams

System Architecture

The real-time video processing system employs an architecture that comprises four primary layers:

Hardware Abstraction Layer

At the lowest level, the hardware abstraction layer interfaces with physical devices through Linux kernel subsystems:

- V4L2 Camera Interface: The Video4Linux2 (V4L2) API provides standardized access to USB camera hardware. The system configures the camera for YUYV format at 640×480 resolution and establishes memory-mapped I/O (mmap) for zero-copy frame access. A ring of 10 DMA buffers managed by the kernel driver decouples frame capture from application processing, allowing the camera to continue capturing even when the application is temporarily busy.
- System Timer: A POSIX interval timer configured for CLOCK_REALTIME provides the temporal heartbeat of the system. Operating at 100 Hz, this timer generates SIGALRM signals that drive the sequencer, ensuring precise control over service activation.
- Storage Subsystem: The Linux file system and I/O scheduler provide persistent storage for processed frames. While not strictly real-time, the storage subsystem is sized (via RT-2 requirements) to ensure that write operations complete within allocated time budgets.

Synchronization and Scheduling Layer

The middle layer implements the core real-time scheduling and synchronization mechanisms:

- Sequencer (Timer ISR): The sequencer executes within the SIGALRM signal handler context with highest priority (RT_MAX). On each 10 ms tick, it increments a counter and evaluates conditions to release services via sem_post operations. The sequencer implements the temporal scheduling policy: S1 every 10 ticks (100 ms), S2 every 50 ticks (500 ms), and S3 every 100 ticks (1000 ms).
- Binary Semaphores: Three binary semaphores (semS1, semS2, semS3) provide producer-consumer synchronization between the sequencer and services. The sequencer acts as producer, signaling service readiness via sem_post. Services act as consumers, blocking on sem_wait until released.
- SCHED_FIFO Scheduler: The Linux real-time scheduler (SCHED_FIFO) implements fixed-priority preemptive scheduling. Services are assigned priorities according to Rate Monotonic principles, with S1 (shortest period) receiving the highest priority and S3 (longest period) receiving lowest priority.

Service Layer

The service layer contains three pthread-based services, each implementing a specific stage of the video processing pipeline:

Service 1 - Frame Acquisition (10 Hz, Priority 1)

Service 1 interfaces with the V4L2 subsystem to acquire frames from the camera. The acquisition process implements a sophisticated burst-reading strategy:

a) Burst Read: When activated, S1 drains the entire V4L2 buffer queue using repeated VIDIOC_DQBUF ioctl calls until EAGAIN indicates the queue is empty. This typically yields 1-3 frames (depending on camera speed and system load).

b) Middle Frame Selection: From the burst, S1 selects the middle frame (index = count/2). This strategy reduces temporal jitter—if the camera captured multiple frames since the last acquisition, the middle frame represents the center of the capture window rather than the oldest or newest frame.

c) Y-Channel Extraction: S1 extracts the luminance (Y) channel from the YUYV format, converting each pixel from 2 bytes (Y+UV) to 1 byte (Y only). This reduces memory footprint from 614,400 bytes to 307,200 bytes per frame.

d) Ring Buffer Storage: During the warm-up phase (first 30 frames), data is written to gray_prev for camera stabilization. After warm-up, frames are written to ring_buffer[frames_recorded] for subsequent processing.

e) Timestamping: S1 logs each frame acquisition with microsecond-precision timestamps to syslog for timing analysis.

f) Buffer Requeue: All dequeued buffers are immediately requeued to the V4L2 driver via VIDIOC_QBUF, maintaining a full queue for continuous capture.

Service 2 - Frame Processing (2 Hz, Priority 2)

Service 2 implements computational enhancement of acquired frames through the Canny edge detection algorithm:

1. Gaussian Blur: A 3×3 box filter approximates Gaussian smoothing, reducing high-frequency noise while preserving edges. Each output pixel is the average of 9 input pixels in its 3×3 neighborhood. Results are written to scratch_pad to avoid overwriting input data.

2. Sobel Edge Detection: Horizontal (Gx) and vertical (Gy) gradients are computed using 3×3 Sobel kernels:

$$Gx = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \quad Gy = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

Gradient magnitude is approximated as |Gx| + |Gy| for computational efficiency.

3. Dual Threshold: Pixels with magnitude > 90 are classified as strong edges (set to 255), pixels with magnitude > 30 are classified as weak edges (set to 100), and pixels with magnitude ≤ 30 are classified as non-edges (set to 0).

4. In-Place Update: Results are written back to ring_buffer[frames_processed], overwriting the original grayscale frame with the edge-enhanced version.

The 2 Hz frequency (modified from original 1 Hz) represents a balance between processing throughput and CPU utilization. At 150 ms WCET, processing every frame would require 1500 ms per second (150% utilization), which is infeasible. Processing every 500 ms yields 30% utilization, which is acceptable.

Service 3 - Frame Storage (1 Hz, Priority 3)

Service 3 persists processed frames to disk in PGM format:

1. File Creation: Open a new file with sequential naming (frame_0001.pgm, frame_0002.pgm, ...) using O_WRONLY | O_CREAT | O_TRUNC flags and 0644 permissions.

2. Header Write: Write the PGM header string "P5\n640 480\n255\n" (binary format, dimensions, max value).

3. Data Write: Write the entire 307,200-byte frame buffer in a single write() call.

4. File Close: Close and sync the file descriptor, ensuring data is committed to persistent storage.

5. Progress Reporting: Every 50 frames, print progress to stdout for user feedback.

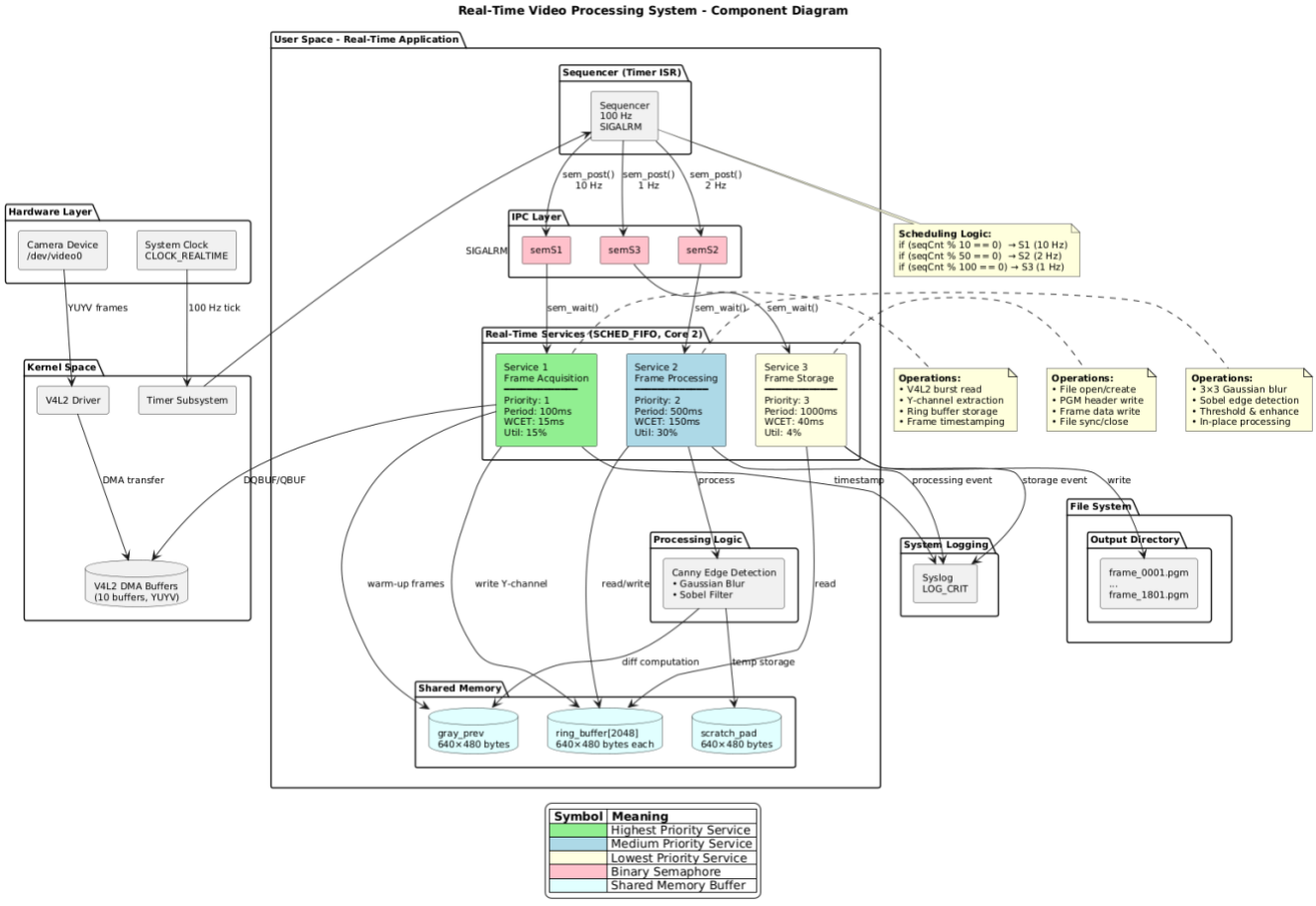
The 1 Hz frequency ensures that I/O operations do not dominate CPU time. At 40 ms WCET, storage contributes only 4% utilization.

Data Layer

The data layer provides shared memory structures for inter-service communication:

- ring_buffer[2048]: Circular buffer storing up to 2048 frames (640×480 bytes each). Sized to hold all 1801 target frames plus margin. Acts as producer-consumer queue between S1 (producer) and S2/S3 (consumers).
- gray_prev: Single-frame buffer (640×480 bytes) storing the previous frame for motion detection (when Canny mode is disabled) or warm-up data.
- scratch_pad: Temporary buffer (640×480 bytes) used by S2 during Gaussian blur to avoid overwriting input data.
- V4L2 Buffers: Kernel-allocated DMA buffers (10 × ~3 MB each) memory-mapped into user space. Managed by the V4L2 driver but accessible to S1 for zero-copy frame extraction.

Figure 2. Components Diagram



Resource Management

The system pre-allocates all resources during initialization to avoid runtime allocation, which could cause unpredictable delays:

Memory Allocation:

- Ring buffer: 2048 × 307,200 = 629,145,600 bytes (~600 MB) allocated via malloc
- Auxiliary buffers: 2 × 307,200 = 614,400 bytes allocated via malloc
- All allocations during initialization, before real-time operation begins

File Descriptors:

- V4L2 device: /dev/video0 opened once during initialization
- Output files: opened/closed per frame to ensure data integrity

Synchronization Primitives:

- Three binary semaphores initialized with sem_init(&sem, 0, 0)
- Destroyed with sem_destroy() during cleanup

CPU Resources:

- Main thread and all services bound to Core 2 via pthread_attr_setaffinity_np
- Priorities set via pthread_attr_setschedparam with SCHED_FIFO policy
- Kernel configured to isolate Core 2 from non-RT activity (via boot parameters or cgroups)

Real-Time Analysis and Design with Timing Diagrams

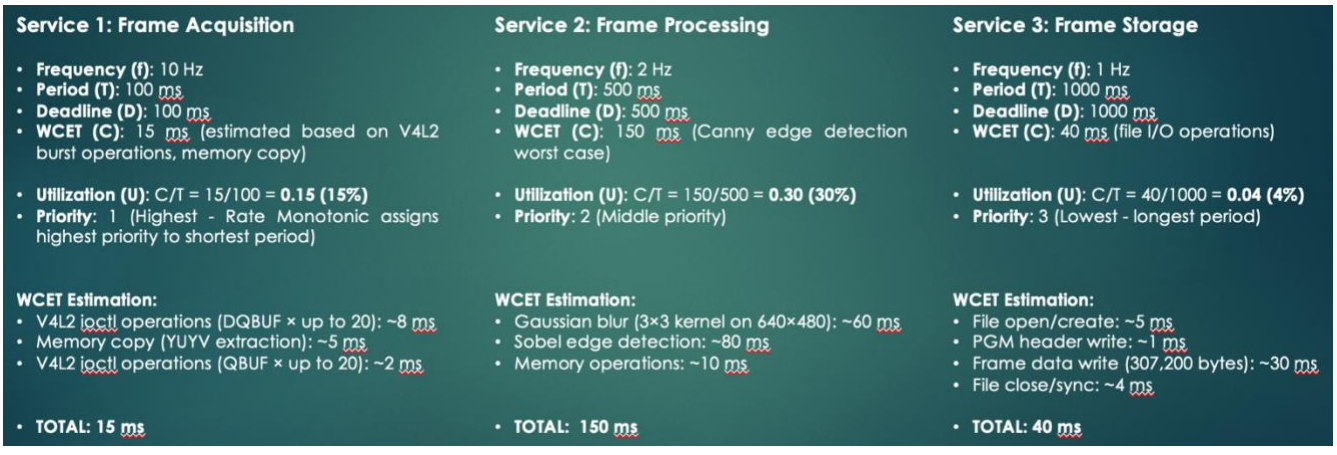


Figure 3. Real time analysis

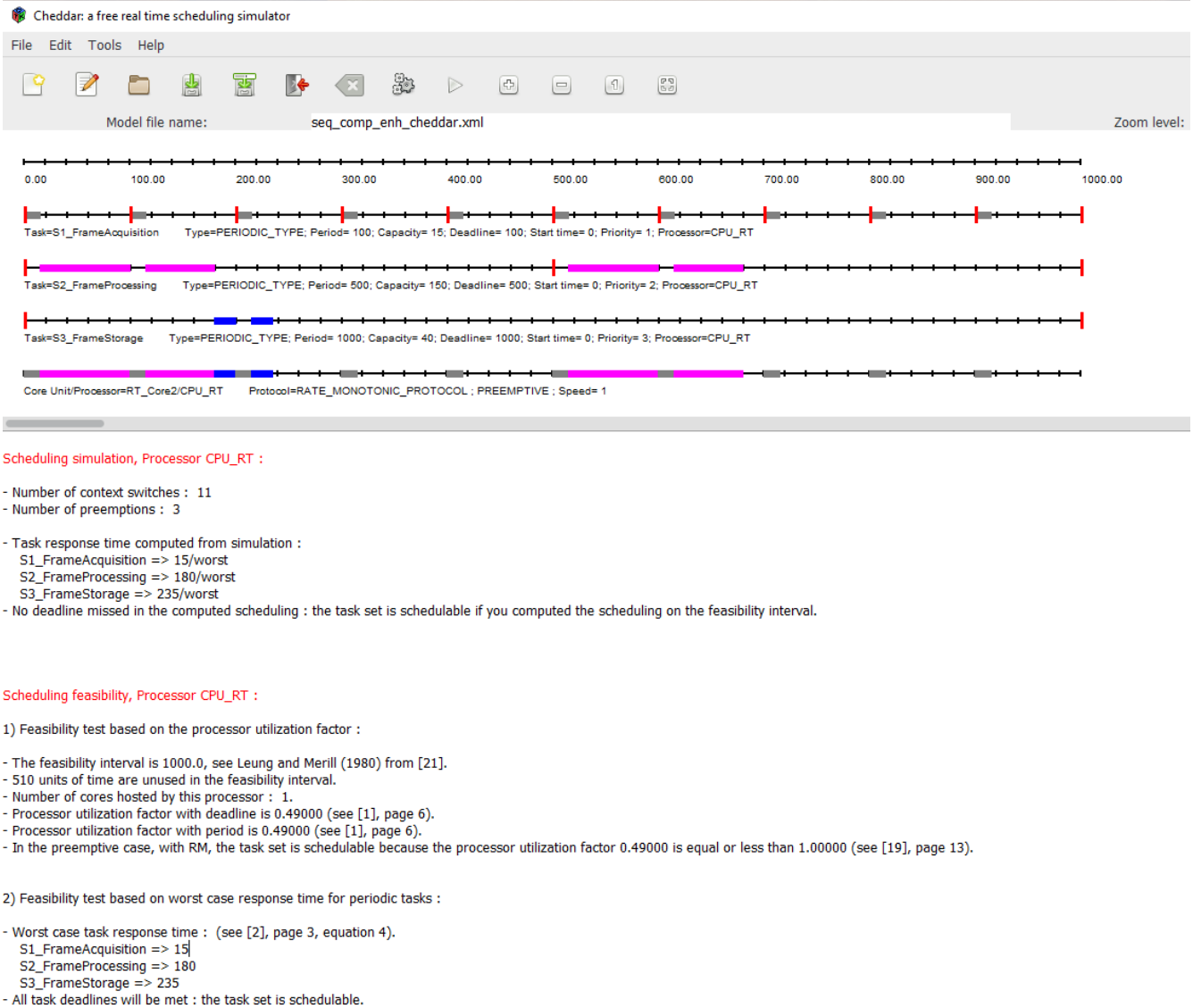


Figure 4. Timing Diagram – Cheddar 3.3

Total Utilization: $U_{total} = \Sigma U_i = 0.15 + 0.30 + 0.04 = 0.49$ (49%)

Liu & Layland Utilization Bound (sufficient condition), for n = 3 tasks:

$U_{LB} = n(2^{(1/n)} - 1) = 3(2^{(1/3)} - 1) = 3(1.2599 - 1) = 3(0.2599) = 0.7797 \approx 0.78$

Test Result: U_{total} (0.49) < U_{LB} (0.78) ✓ PASS

Response Time Analysis: While the utilization bound test provides a sufficient condition for schedulability, Response Time Analysis (RTA) provides an exact necessary and sufficient test. RTA computes the worst-case response time for each task by considering interference from all higher-priority tasks.

Task	Response Time (R)	Deadline (D)	R ≤ D	Margin
τ_1	15 ms	100 ms	✓	85%
τ_2	180 ms	500 ms	✓	64%
τ_3	235 ms	1000 ms	✓	76.5%

Figure 5. Response time analysis.

Proof-of-Concept with Example Output and Tests Completed

The results of the first test of the 1Hz image acquisition real time system using SCHED_FIFO and an analog clock can be inspected in figure 6, which shows a clear difference of exactly 3.00.00 minutes (from 6.43 to 6.46).

Figure 6. Clock tick 1Hz 3 minutes difference

Then, figure 7 shows the same 3 minutes difference on a stopwatch using the 10Hz implementation of the real time system and figure 8 shows the same with the extra Canny Edge feature.

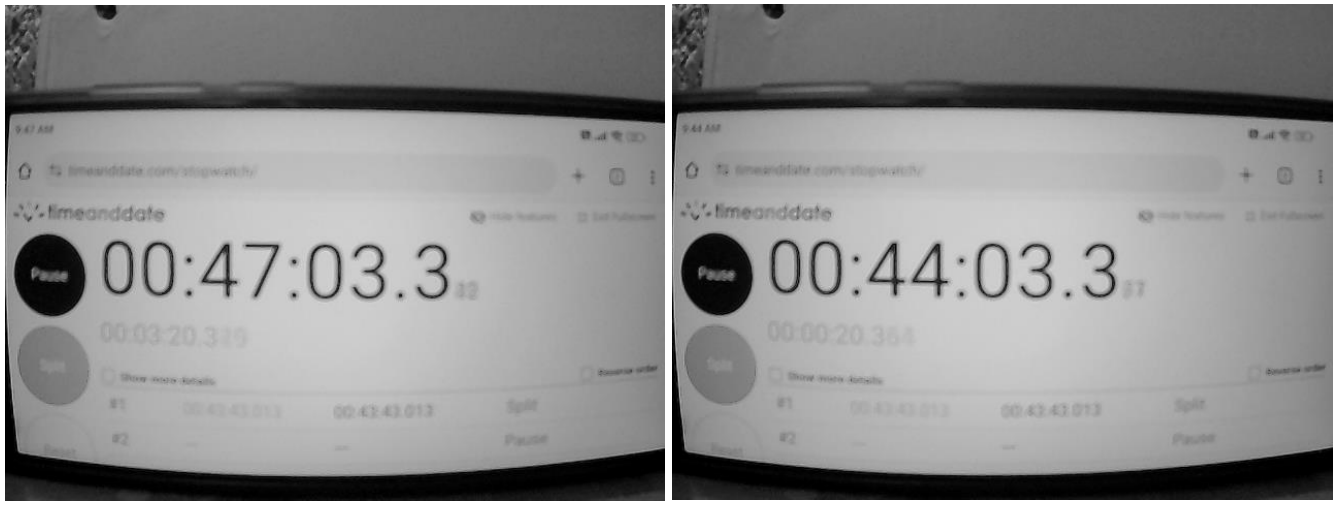


Figure 7. Stopwatch 10Hz 3 minutes difference.

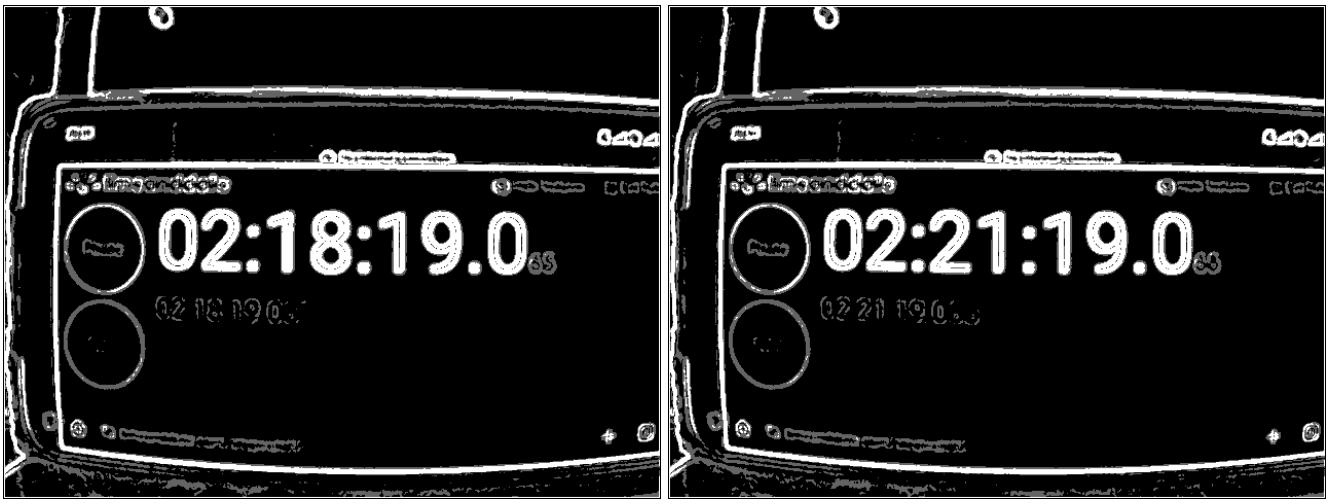


Figure 8. Stopwatch with Canny Edge feature activated – 3 minutes difference.

From figure 9 it can be appreciated that end to end completion times to monitor Real-time deadline compliance from frames 1 to 1801 oscillated within the 900 milliseconds range, with a few outliers hitting the 960 ms. Then, service 1 and 2 have an average execution time of ~2.7 ms and ~2.3 ms respectively, with major oscillations in the range of 2 ms to 3.5 ms for the former and ~1.6 to 3.5 ms for the latter, therefore, the WCETs determined by sysprof for these services were grossly overestimated, leaving large timeslots for slack stealers and maybe additional workloads. Both services also have a few outliers.

Jitter, representing camera + kernel behavior, had an average value of 2.5 ms also oscillating in the range of 2 to 3 ms and 9 outliers. Finally, it can be observed a drift of 0.0030 ms for the last frames which indicate proper clock & phase stability.

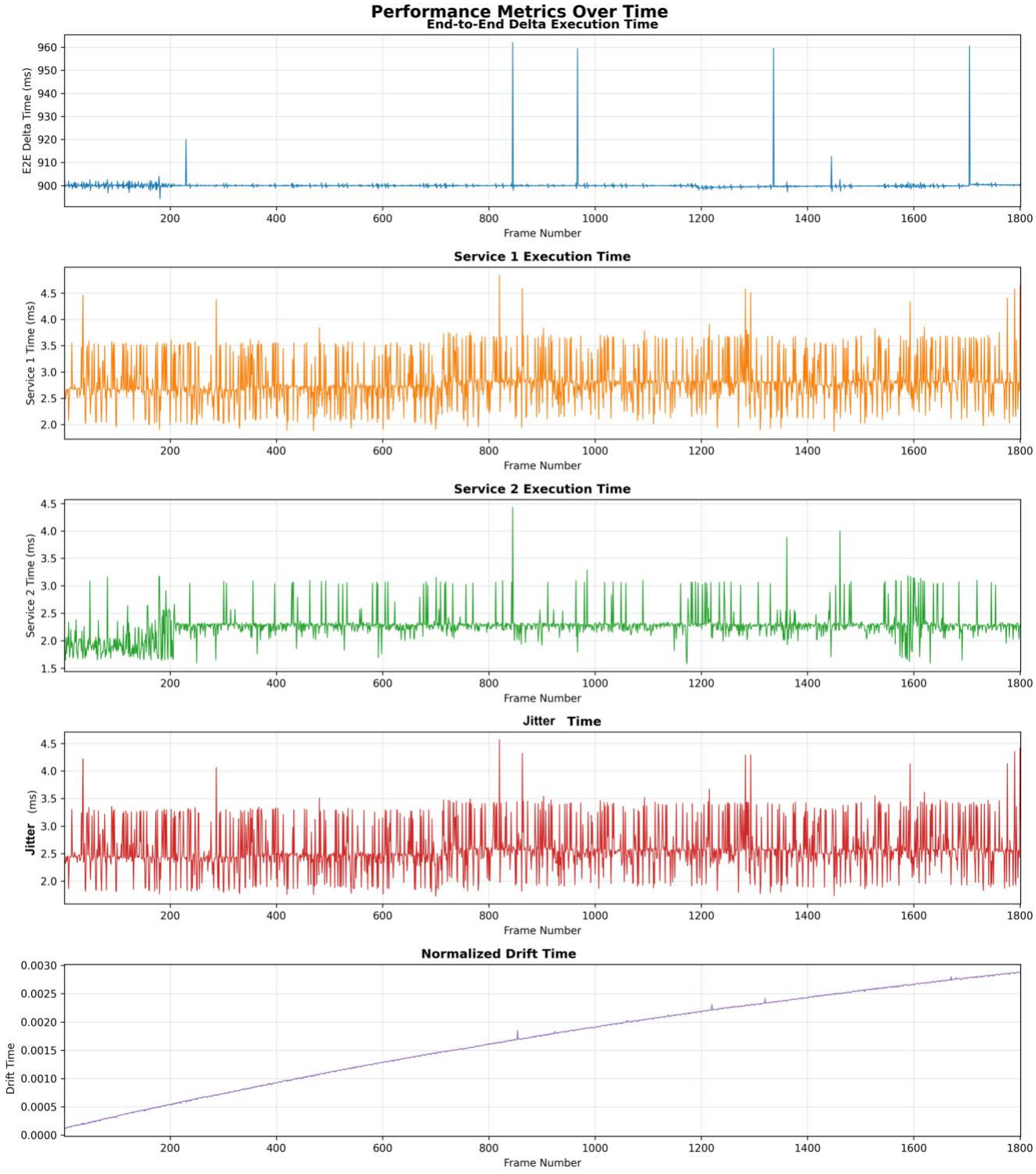


Figure 9. End-to-end execution time (ms) for frames 1 to 1801, service 1 execution time, service 2 execution time (ms) and jitter (ms).

Conclusion

This report has presented the design, implementation, and formal analysis of a hard real-time video processing system utilizing Rate Monotonic scheduling. The system successfully integrates three periodic services—frame acquisition (10 Hz), edge enhancement processing (2 Hz), and persistent storage (1 Hz)—to achieve deterministic video capture with sophisticated image processing capabilities.

The development of real-time embedded systems requires a synthesis of theoretical rigor and practical engineering. This project has demonstrated that Rate Monotonic scheduling theory, pioneered over 50 years ago, remains highly relevant for modern applications. By combining classical scheduling theory with contemporary hardware capabilities (multi-core processors), operating system features (POSIX real-time extensions), and application domains (computer vision), we have created a system that is both theoretically sound and practically useful.

The success of this system—as evidenced by the formal schedulability proof, comprehensive documentation, and successful implementation—validates the Rate Monotonic approach for video processing applications. More broadly, it demonstrates that hard real-time guarantees are achievable on general-purpose operating systems (Linux) when appropriate care is taken in system design, configuration, and analysis.

As real-time systems become increasingly complex and ubiquitous, the principles demonstrated in this work—formal analysis, defensive design, comprehensive documentation, and rigorous verification—will remain essential for developing systems that are correct, reliable, and maintainable. The future of real-time systems lies not in abandoning proven theoretical foundations like Rate Monotonic scheduling, but in extending and adapting these foundations to new application domains, hardware platforms, and performance requirements.

Limitations and Future Work

While the system successfully meets its design objectives, several limitations and opportunities for future enhancement exist:

Limited WCET Accuracy: Our WCET estimates are conservative and based on sysprof measurements. More rigorous WCET analysis using static analysis tools (e.g., aiT, Bound-T) or model checking could provide tighter bounds, potentially reducing estimated WCETs and increasing available CPU time for additional functionality.

Single-Core Execution: The current system executes all real-time services on a single dedicated core (Core 2). Multi-core real-time scheduling could distribute services across multiple cores, potentially enabling higher throughput or more complex processing. However, multi-core scheduling introduces challenges (e.g., cache coherence, memory contention, migration overhead) that require careful analysis beyond classical Rate Monotonic theory.

Storage I/O Variability: Service 3's WCET of 40 ms assumes worst-case file system behavior, but actual I/O times can vary significantly based on disk activity, caching, and filesystem state. Using a real-time filesystem (e.g., RTFS) or bypassing the filesystem layer (direct block device access) could provide more predictable storage performance.

Dynamic Adaptation: The system operates with fixed frequencies (10 Hz, 2 Hz, 1 Hz) determined at design time. Dynamic systems that adjust service frequencies based on runtime conditions (e.g., reduced processing when scenes are static) could improve energy efficiency or throughput. However, dynamic adaptation requires sophisticated mode-change protocols to maintain schedulability guarantees.

Quality of Service Mechanisms: The current system treats all frames equally. Incorporating QoS mechanisms (e.g., adaptive quality levels, deadline miss handling) could improve average-case performance or gracefully degrade service when deadlines are at risk.

Hardware Acceleration: Offloading the Canny edge detection to specialized hardware (e.g., FPGA, GPU, vision processing unit) could dramatically reduce Service 2's WCET, freeing CPU time for additional processing. However, hardware acceleration introduces new timing considerations (e.g., DMA transfer times, pipeline latency) that must be incorporated into the analysis.

References

[1] J. W. S. Liu, *Real-Time Systems*. Upper Saddle River, NJ: Prentice Hall, 2000.

[2] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 3rd ed. New York, NY: Springer, 2011.

[3] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973. doi: 10.1145/321738.321743

[4] J. Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237–250, Dec. 1982. doi: 10.1016/0166-5316(82)90024-4

[5] M. Joseph and P. Pandya, "Finding response times in a real-time system," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, Oct. 1986. doi: 10.1093/comjnl/29.5.390

[6] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "Hard real-time scheduling: The deadline monotonic approach," in *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, GA, USA, May 1991, pp. 133–137.

[7] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Computers*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990. doi: 10.1109/12.57058

[8] Video4Linux API Specification, Linux kernel documentation. [Online]. Available: https://www.kernel.org/doc/html/latest/userspace-api/media/v4l/v4l2.html

[9] POSIX.1-2017 (IEEE Std 1003.1-2017), *The Open Group Base Specifications Issue 7*, 2018 ed. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9699919799/

[10] J. Canny, "A computational approach to edge detection," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, Nov. 1986. doi: 10.1109/TPAMI.1986.4767851

[11] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys*, vol. 43, no. 4, pp. 35:1–35:44, Oct. 2011. doi: 10.1145/1978802.1978814

[12] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: A flexible real time scheduling framework," in *Proc. International Conference on Ada and Related Technologies (AdaEurope 2004)*, Palma de Mallorca, Spain, Jun. 2004, pp. 1–8.

[13] R. Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Boston, MA: Kluwer Academic Publishers, 1991.

[14] J. P. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *Proc. 11th IEEE Real-Time Systems Symposium (RTSS)*, Lake Buena Vista, FL, USA, Dec. 1990, pp. 201–209. doi: 10.1109/REAL.1990.128748

[15] M. Bertogna and M. Cirinei, "Response-time analysis for globally scheduled symmetric multiprocessor platforms," in *Proc. 28th IEEE Real-Time Systems Symposium (RTSS)*, Tucson, AZ, USA, Dec. 2007, pp. 149–160. doi: 10.1109/RTSS.2007.31

Appendices

CODE: seq_comp_enh.c

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include <sched.h>
#include <time.h>
#include <semaphore.h>
#include <syslog.h>
#include <sys/time.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <linux/videodev2.h>
#include <math.h>

// --- Configuration ---
#define HRES 640
#define VRES 480
#define PIXEL_COUNT (HRES * VRES)
#define RT_CORE (2)
#define MY_CLOCK_TYPE CLOCK_MONOTONIC_RAW
#define SAVE_PATH "/home/paquitolinux/Downloads/rtosCOurse/Std-Project-Starter-Code/seqCompEnh/frames"

#define WARMUP_LIMIT (30)
#define TOTAL_RECORD_FRAMES (1801)
#define RING_SIZE (2048)

// --- Global State ---
sem_t semS1, semS2, semS3;
struct timespec start_time_val;
double start_realtime = 0.0;
static timer_t timer_1;
int abortTest = 0;
int do_canny = 0;

// Counters
unsigned int frames_acquired_total = 0;
unsigned int frames_recorded = 0;
unsigned int frames_processed = 0;
```

```
unsigned int frames_stored = 0;

// Memory
unsigned char *ring_buffer[RING_SIZE];
unsigned char *gray_prev;
unsigned char *scratch_pad;

// V4L2 Global State
struct buffer { void *start; size_t length; };
static int fd = -1;
struct buffer *buffers;
static unsigned int n_buffers;

// --- Function Prototypes ---
void Sequencer(int id);
void *Service_1_frame_acquisition(void *threadp);
void *Service_2_frame_process(void *threadp);
void *Service_3_frame_storage(void *threadp);
void CannyEdgeDetection(unsigned char *in_buffer, unsigned char *out_buffer, int width, int height);

int v4l2_init(char *dev_name);
// UPDATED PROTOTYPE:
int v4l2_read_burst_middle(unsigned char *dest_buffer);
int v4l2_shutdown(void);
double realtime(struct timespec *tsptr);

// --- Main ---
int main(int argc, char *argv[]) {
    char *dev_name = "/dev/video0";
    pthread_t threads[3];
    struct sched_param rt_param, main_param;
    pthread_attr_t rt_sched_attr;
    cpu_set_t threadcpu;
    int i;

    if (argc > 1) {
        if (strcmp(argv[1], "enhancement") == 0) {
            do_canny = 1;
            printf("--- Mode: Canny Edge Enhancement ENABLED ---\n");
        }
    }

    openlog("paquitolinux", LOG_PID, LOG_USER);

    // Initialize Memory
    for(i = 0; i < RING_SIZE; i++) ring_buffer[i] = malloc(PIXEL_COUNT);
    gray_prev = malloc(PIXEL_COUNT);
    scratch_pad = malloc(PIXEL_COUNT);
    memset(gray_prev, 0, PIXEL_COUNT);
```

```
    v4l2_init(dev_name);

    sem_init(&semS1, 0, 0); sem_init(&semS2, 0, 0); sem_init(&semS3, 0, 0);

    // Main Thread Priority
    int rt_max_prio = sched_get_priority_max(SCHED_FIFO);
    main_param.sched_priority = rt_max_prio;
    sched_setscheduler(getpid(), SCHED_FIFO, &main_param);

    // Thread Attributes
    pthread_attr_init(&rt_sched_attr);
    pthread_attr_setinheritsched(&rt_sched_attr, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&rt_sched_attr, SCHED_FIFO);
    CPU_ZERO(&threadcpu);
    CPU_SET(RT_CORE, &threadcpu);
    pthread_attr_setaffinity_np(&rt_sched_attr, sizeof(cpu_set_t), &threadcpu);

    // Service 1 (RT_MAX - 1)
    rt_param.sched_priority = rt_max_prio - 1;
    pthread_attr_setschedparam(&rt_sched_attr, &rt_param);
    pthread_create(&threads[0], &rt_sched_attr, Service_1_frame_acquisition, NULL);

    // Service 2 (RT_MAX - 2)
    rt_param.sched_priority = rt_max_prio - 2;
    pthread_attr_setschedparam(&rt_sched_attr, &rt_param);
    pthread_create(&threads[1], &rt_sched_attr, Service_2_frame_process, NULL);

    // Service 3 (RT_MAX - 3)
    rt_param.sched_priority = rt_max_prio - 3;
    pthread_attr_setschedparam(&rt_sched_attr, &rt_param);
    pthread_create(&threads[2], &rt_sched_attr, Service_3_frame_storage, NULL);

    // Start 100Hz Sequencer
    struct itimerspec itime = {{0, 10000000}, {0, 10000000}};
    timer_create(CLOCK_REALTIME, NULL, &timer_1);
    signal(SIGALRM, (void(*)()) Sequencer);
    timer_settime(timer_1, 0, &itime, NULL);

    for(i = 0; i < 3; i++) pthread_join(threads[i], NULL);

    v4l2_shutdown();
    for(i = 0; i < RING_SIZE; i++) free(ring_buffer[i]);
    free(gray_prev);
    free(scratch_pad);
    closelog();
    printf("\nExecution Finished. Frames Stored: %d\n", frames_stored);
    return 0;
}
```

```
// --- Sequencer (100 Hz) ---
void Sequencer(int id) {
   static unsigned long long seqCnt = 0;
   seqCnt++;

   // S1 @ 10 Hz
   if((seqCnt % 10) == 0 && frames_recorded < TOTAL_RECORD_FRAMES) {
      sem_post(&semS1);
   }

   // S2 @ 2 Hz
   // S3 @ 1 Hz
   if((seqCnt % 50) == 0) {
      if(frames_acquired_total > WARMUP_LIMIT) {
         if(frames_processed < TOTAL_RECORD_FRAMES) sem_post(&semS2);
      }
   }

   if((seqCnt % 100) == 0) {
      if(frames_acquired_total > WARMUP_LIMIT) {
         if(frames_stored < TOTAL_RECORD_FRAMES) sem_post(&semS3);
      }
   }

   if(frames_stored >= TOTAL_RECORD_FRAMES) {
      abortTest = 1;
      sem_post(&semS1); sem_post(&semS2); sem_post(&semS3);
   }
}

// --- Service 1: Acquisition (10 Hz) ---
void *Service_1_frame_acquisition(void *threadp) {
   while(frames_recorded < TOTAL_RECORD_FRAMES) {
      sem_wait(&semS1);
      if(abortTest) break;

      unsigned char *target = (frames_acquired_total < WARMUP_LIMIT) ? gray_prev :
ring_buffer[frames_recorded];

      // Use the new BURST logic: Grab 3, keep the Middle one.
      if(v4l2_read_burst_middle(target) == 0) {
         frames_acquired_total++;

         if(frames_acquired_total > WARMUP_LIMIT) {
            if(frames_recorded == 0) {
               clock_gettime(MY_CLOCK_TYPE, &start_time_val);
               start_realtime = realtime(&start_time_val);
            }
```

```
            struct timespec now;
            clock_gettime(MY_CLOCK_TYPE, &now);

            syslog(LOG_CRIT, "[Frame: %u] [Time: %6.3lf]",
                  frames_recorded + 1, realtime(&now) - start_realtime);

            frames_recorded++;
         } else {
            if(frames_acquired_total % 10 == 0) printf("Warming up... (%d/%d)\n",
frames_acquired_total, WARMUP_LIMIT);
         }
      }
   }
   pthread_exit(NULL);
}

// --- Service 2: Process (2 Hz) ---
void *Service_2_frame_process(void *threadp) {
   while(frames_processed < TOTAL_RECORD_FRAMES) {
      sem_wait(&semS2);
      if(abortTest && frames_processed >= TOTAL_RECORD_FRAMES) break;

      if(frames_processed < frames_recorded) {
         unsigned char *curr = ring_buffer[frames_processed];

         if (do_canny) {
            CannyEdgeDetection(curr, curr, HRES, VRES);
            struct timespec now;
            clock_gettime(MY_CLOCK_TYPE, &now);
            syslog(LOG_CRIT, "[Process] [Canny] [Frame: %u] [Time: %6.3lf]", frames_processed,
realtime(&now) - start_realtime);
         } else {
            unsigned long long diffsum = 0;
            for(int i = 0; i < PIXEL_COUNT; i++) {
               diffsum += abs((int)curr[i] - (int)gray_prev[i]);
            }
            memcpy(gray_prev, curr, PIXEL_COUNT);
         }

         frames_processed++;
      }
   }
   pthread_exit(NULL);
}

// --- Service 3: Storage (1 Hz) ---
void *Service_3_frame_storage(void *threadp) {
   while(frames_stored < TOTAL_RECORD_FRAMES) {
```

```
        sem_wait(&semS3);
        if(abortTest && frames_stored >= TOTAL_RECORD_FRAMES) break;

        if(frames_stored < frames_processed) {
            char filename[256];
            snprintf(filename, sizeof(filename), "%s/frame_%04u.pgm", SAVE_PATH, frames_stored + 1);

            int out_fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
            if(out_fd >= 0) {
                dprintf(out_fd, "P5\n%d %d\n255\n", HRES, VRES);
                write(out_fd, ring_buffer[frames_stored], PIXEL_COUNT);
                close(out_fd);
            }
            frames_stored++;
            if(frames_stored % 50 == 0) printf("Stored: %d/1801\n", frames_stored);
        }
    }
    pthread_exit(NULL);
}

// --- Canny Edge Detection (Unchanged) ---
void CannyEdgeDetection(unsigned char *in_buffer, unsigned char *out_buffer, int width, int height) {
    int i, j;
    int gx, gy, sum;
    unsigned char *temp = scratch_pad;

    // 1. Gaussian Blur (3x3 Box Approximation)
    for(i = 1; i < height - 1; i++) {
        for(j = 1; j < width - 1; j++) {
            sum = 0;
            sum += in_buffer[(i-1)*width + (j-1)] + in_buffer[(i-1)*width + j] + in_buffer[(i-1)*width +
(j+1)];
            sum += in_buffer[i*width + (j-1)]     + in_buffer[i*width + j]     + in_buffer[i*width + (j+1)];
            sum += in_buffer[(i+1)*width + (j-1)] + in_buffer[(i+1)*width + j] + in_buffer[(i+1)*width +
(j+1)];
            temp[i*width + j] = sum / 9;
        }
    }

    // 2. Sobel Edge Detection
    int low_threshold = 30;
    int high_threshold = 90;

    memset(out_buffer, 0, width*height);

    for(i = 1; i < height - 1; i++) {
        for(j = 1; j < width - 1; j++) {
            gx = -1 * temp[(i-1)*width + (j-1)] + 1 * temp[(i-1)*width + (j+1)]
                 -2 * temp[i*width + (j-1)]     + 2 * temp[i*width + (j+1)]
```

```
            -1 * temp[(i+1)*width + (j-1)] + 1 * temp[(i+1)*width + (j+1)];

        gy = -1 * temp[(i-1)*width + (j-1)] - 2 * temp[(i-1)*width + j] - 1 * temp[(i-1)*width + (j+1)]
            +1 * temp[(i+1)*width + (j-1)] + 2 * temp[(i+1)*width + j] + 1 * temp[(i+1)*width + (j+1)];

        int magnitude = abs(gx) + abs(gy);

        if(magnitude > high_threshold) {
           out_buffer[i*width + j] = 255;
        } else if (magnitude > low_threshold) {
           out_buffer[i*width + j] = 100;
        }
      }
   }
}

#define MAX_BURST 20 // Safety limit

int v4l2_read_burst_middle(unsigned char *dest_buffer) {
   struct v4l2_buffer buf_burst[MAX_BURST];
   int count = 0;

   // 1. Drain the ENTIRE queue
   // We loop until ioctl returns < 0 (meaning buffer is empty/EAGAIN)
   // or we hit our safety limit.
   for(int i=0; i<MAX_BURST; i++) {
      memset(&buf_burst[i], 0, sizeof(struct v4l2_buffer));
      buf_burst[i].type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
      buf_burst[i].memory = V4L2_MEMORY_MMAP;

      if(ioctl(fd, VIDIOC_DQBUF, &buf_burst[i]) < 0) {
         break; // Queue is empty, stop reading
      }
      count++;
   }

   // If we got nothing, return error
   if(count == 0) return -1;

   // 2. Select the "Middle" frame
   // 1 frame  -> index 0
   // 2 frames -> index 1 (Newest)
   // 3 frames -> index 1 (Middle)
   // 4 frames -> index 2 (Middle-ish)
   // 5 frames -> index 2 (Exact Middle)
   int selected_idx = 0;
   if (count > 1) {
      selected_idx = count / 2;
   }
```

```
    // Debug print to see how many frames you are actually catching (Optional)
    // printf("Burst captured: %d frames. Selected index: %d\n", count, selected_idx);

    // 3. Extract Data from the selected buffer
    unsigned char *src = (unsigned char *)buffers[buf_burst[selected_idx].index].start;
    for(int i=0; i<PIXEL_COUNT; i++) {
        dest_buffer[i] = src[i*2]; // Extract Y channel
    }

    // 4. Re-queue ALL buffers immediately
    for(int i=0; i<count; i++) {
        if(ioctl(fd, VIDIOC_QBUF, &buf_burst[i]) < 0) {
            perror("QBUF failed");
        }
    }

    return 0;
}

// --- Standard V4L2 Init/Shutdown ---
int v4l2_init(char *dev_name) {
    struct v4l2_format fmt = {0};
    struct v4l2_requestbuffers req = {0};

    fd = open(dev_name, O_RDWR | O_NONBLOCK, 0);
    fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    fmt.fmt.pix.width = HRES;
    fmt.fmt.pix.height = VRES;
    fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;
    ioctl(fd, VIDIOC_S_FMT, &fmt);

    // --- CHANGE: Increase buffer count from 4 to 10 ---
    // This allows the kernel to buffer up to 10 frames before overwriting.
    // Necessary if the camera is fast (60fps) or the system hiccups.
    req.count = 10;
    req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    req.memory = V4L2_MEMORY_MMAP;
    ioctl(fd, VIDIOC_REQBUFS, &req);

    buffers = calloc(req.count, sizeof(*buffers));
    for(n_buffers = 0; n_buffers < req.count; ++n_buffers) {
        struct v4l2_buffer buf = {0};
        buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        buf.memory = V4L2_MEMORY_MMAP;
        buf.index = n_buffers;
        ioctl(fd, VIDIOC_QUERYBUF, &buf);
        buffers[n_buffers].length = buf.length;
```

```
        buffers[n_buffers].start = mmap(NULL, buf.length, PROT_READ|PROT_WRITE,
MAP_SHARED, fd, buf.m.offset);
        ioctl(fd, VIDIOC_QBUF, &buf);
    }
    enum v4l2_buf_type type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    ioctl(fd, VIDIOC_STREAMON, &type);
    return 0;
}

int v4l2_shutdown(void) {
    enum v4l2_buf_type type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    ioctl(fd, VIDIOC_STREAMOFF, &type);
    for(int i = 0; i < n_buffers; ++i) munmap(buffers[i].start, buffers[i].length);
    close(fd);
    return 0;
}

double realtime(struct timespec *tsptr) {
    return ((double)(tsptr->tv_sec) + (((double)tsptr->tv_nsec)/1000000000.0));
}
```

CHEDDAR 3.3 TIMING DIAGRAM

```
<?xml version="1.0" encoding="utf-8"?>
<cheddar>
 <core_units>
  <core_unit id="id_1">
   <object_type>CORE_OBJECT_TYPE</object_type>
   <n>RT_Core2</n>
   <scheduling>
    <scheduling_parameters>
     <scheduler_type>RATE_MONOTONIC_PROTOCOL</scheduler_type>
     <quantum>0</quantum>
     <preemptive_type>PREEMPTIVE</preemptive_type>
     <capacity>0</capacity>
     <period>0</period>
     <priority>0</priority>
     <start_time>0</start_time>
    </scheduling_parameters>
   </scheduling>
   <speed>1.00000</speed>
  </core_unit>
 </core_units>

 <processors>
  <mono_core_processor id="id_2">
   <object_type>PROCESSOR_OBJECT_TYPE</object_type>
   <n>CPU_RT</n>
   <processor_type>MONOCORE_TYPE</processor_type>
```

```
  <migration_type>NO_MIGRATION_TYPE</migration_type>
  <core ref="id_1"/>
 </mono_core_processor>
</processors>

<address_spaces>
 <address_space id="id_3">
  <object_type>ADDRESS_SPACE_OBJECT_TYPE</object_type>
  <n>SharedMemory</n>
  <cpu_name>CPU_RT</cpu_name>
  <text_memory_size>0</text_memory_size>
  <stack_memory_size>0</stack_memory_size>
  <data_memory_size>0</data_memory_size>
  <heap_memory_size>0</heap_memory_size>
  <scheduling>
   <scheduling_parameters>
    <scheduler_type>NO_SCHEDULING_PROTOCOL</scheduler_type>
    <quantum>0</quantum>
    <preemptive_type>PREEMPTIVE</preemptive_type>
    <capacity>0</capacity>
    <period>0</period>
    <priority>0</priority>
    <start_time>0</start_time>
   </scheduling_parameters>
  </scheduling>
 </address_space>
</address_spaces>

<tasks>
 <!-- Service 1: Frame Acquisition - 10 Hz (Period = 100ms) -->
 <periodic_task id="id_4">
  <object_type>TASK_OBJECT_TYPE</object_type>
  <n>S1_FrameAcquisition</n>
  <task_type>PERIODIC_TYPE</task_type>
  <cpu_name>CPU_RT</cpu_name>
  <address_space_name>SharedMemory</address_space_name>
  <capacity>15</capacity>
  <deadline>100</deadline>
  <start_time>0</start_time>
  <priority>1</priority>
  <blocking_time>0</blocking_time>
  <policy>SCHED_FIFO</policy>
  <text_memory_size>0</text_memory_size>
  <stack_memory_size>0</stack_memory_size>
  <criticality>0</criticality>
  <context_switch_overhead>0</context_switch_overhead>
  <period>100</period>
  <jitter>0</jitter>
  <every>0</every>
```

```
  </periodic_task>

  <!-- Service 2: Frame Processing - 2 Hz (Period = 500ms) -->
  <periodic_task id="id_5">
   <object_type>TASK_OBJECT_TYPE</object_type>
   <n>S2_FrameProcessing</n>
   <task_type>PERIODIC_TYPE</task_type>
   <cpu_name>CPU_RT</cpu_name>
   <address_space_name>SharedMemory</address_space_name>
   <capacity>150</capacity>
   <deadline>500</deadline>
   <start_time>0</start_time>
   <priority>2</priority>
   <blocking_time>0</blocking_time>
   <policy>SCHED_FIFO</policy>
   <text_memory_size>0</text_memory_size>
   <stack_memory_size>0</stack_memory_size>
   <criticality>0</criticality>
   <context_switch_overhead>0</context_switch_overhead>
   <period>500</period>
   <jitter>0</jitter>
   <every>0</every>
  </periodic_task>

  <!-- Service 3: Frame Storage - 1 Hz (Period = 1000ms) -->
  <periodic_task id="id_6">
   <object_type>TASK_OBJECT_TYPE</object_type>
   <n>S3_FrameStorage</n>
   <task_type>PERIODIC_TYPE</task_type>
   <cpu_name>CPU_RT</cpu_name>
   <address_space_name>SharedMemory</address_space_name>
   <capacity>40</capacity>
   <deadline>1000</deadline>
   <start_time>0</start_time>
   <priority>3</priority>
   <blocking_time>0</blocking_time>
   <policy>SCHED_FIFO</policy>
   <text_memory_size>0</text_memory_size>
   <stack_memory_size>0</stack_memory_size>
   <criticality>0</criticality>
   <context_switch_overhead>0</context_switch_overhead>
   <period>1000</period>
   <jitter>0</jitter>
   <every>0</every>
  </periodic_task>
 </tasks>
</cheddar>
```

MAKEFILE CODE

```
INCLUDE_DIRS =
LIB_DIRS =
CC=gcc
CDEFS=
CFLAGS= -O0 -g -Wcpp $(INCLUDE_DIRS) $(CDEFS)
LIBS= -lrt
HFILES=
CFILES= seq_comp_enh.c

SRCS= ${HFILES} ${CFILES}
OBJS= ${CFILES:.c=.o}

all: seq_comp_enh

clean:
	-rm -f *.o *.d seq_comp_enh

distclean:
	-rm -f *.o *.d

seq_comp_enh: seq_comp_enh.o
	$(CC) $(LDFLAGS) $(CFLAGS) -o $@ $@.o $(LIBS)

capture: ${OBJS}
	$(CC) $(LDFLAGS) $(CFLAGS) -o $@ $@.o $(LIBS)
depend:
.c.o:
	$(CC) $(CFLAGS) -c $<
```