ECEN 5623
University of Colorado Boulder

Exercise 1 Report
Real-Time Embedded Systems

Ali Sulehria

February 2025

Contents

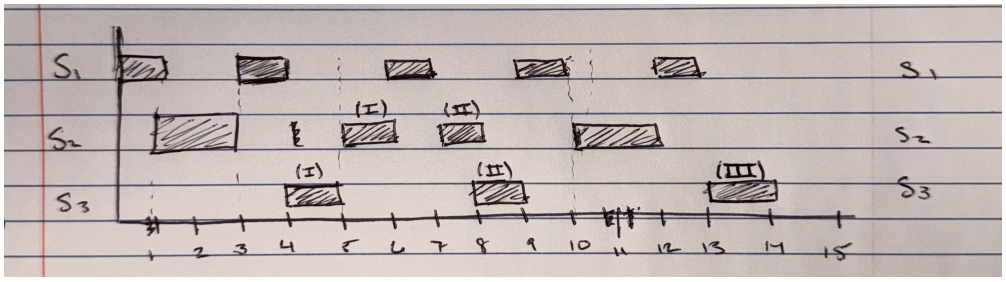

Figure 1: A schedule for the service set in Problem 1.

1 Scheduling and Feasibility

(a) See Figure 1

(b) The schedule is feasible, as each task completes in its allotted period and all tasks are completed before the hyperperiod ends. It is safe in normal to ideal conditions, but only to a point - it has a single millisecond of overhead per hyperperiod, so would not deal with anywhere near a worst-case scenario without exceeding safe parameters.

(c) The total utilization is $\frac{14}{15}$, or 93.33%.

2 Apollo 11's Overload

The Apollo 11 mission's Lunar Module (LM) notably encountered hitches in its landing sequence. The LM's computer, developed by a team at MIT including Margaret Hamilton, processed everything from engine burn duration to sextant readings. At some point in the landing sequence, though, a misconfiguration of the radar switches meant that the computer began processing faulty data which blocked mission-critical tasks. Alarms sounded to alert the Apollo crew to the potential failure. However, one of the most well-tested and thought out capabilities of the system was its restart function, which would almost immediately purge the unnecessary tasks while restarting critical ones as close to their last active state as possible. The restart was able to flush the radar processes from the system, allowing for the LM to land successfully.

(a-c) This may have violated rate monotonic policy because of the influx of many tasks which likely had shorter periods as they were simpler calculations. They could have also gained high priority by unexpectedly interrupting a periodic task set and being elevated because the system assumed they were initialization or failure recovery tasks. Either way, these tasks were assigned higher priority and likely exceeded the least upper bound advised in the Liu & Layland paper, promting an alarm for concerning system load levels. To avoid this, the system should keep utilization below a certain threshold utilization based on the number of tasks being scheduled. The equation for this least upper bound (LUB) of the utilization factor U with m tasks is

$$U = \sum_{i=1}^{m} \frac{C_i}{T_i} = m(2^{\frac{1}{m}} - 1).$$

This is seen in Figure 2, where the LUB converges to just above 69% utilization as $m \to \infty$. This means that a margin of almost 31% is advisable for a system with many tasks $m > 10$.

(d) Liu and Layland assume 1) that tasks have periodic deadlines with constant intervals, 2) that tasks may not depend on other tasks' requests, and 3) that each task's runtime is constant each time it is completed. I am still confused that, as discussed, aperiodic tasks have high priority

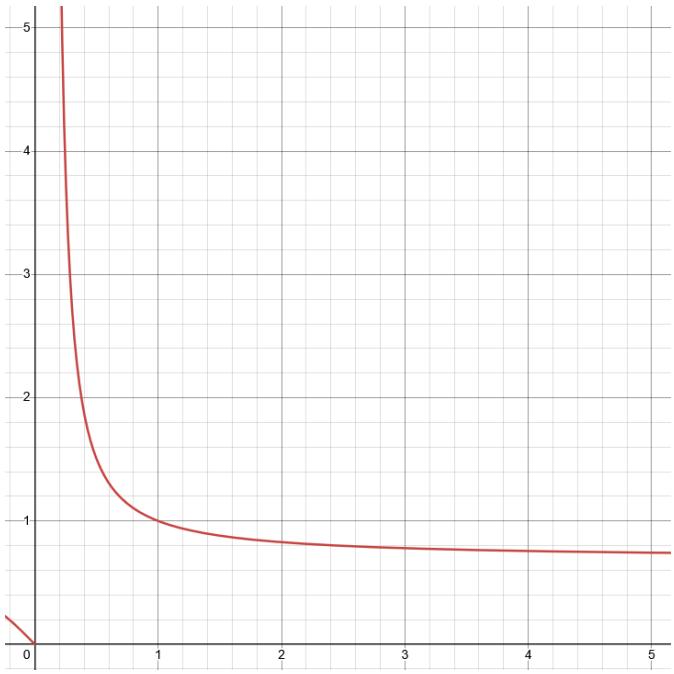

Figure 2: A graph of the law for least upper bound (LUB), per Liu and Layland. Horizontal axis represents increasing task count, while the vertical indicates processor utilization U. U decays asymptotically to just above 0.69.

when requested. Is the aperiodicity a function of a sudden entrance into the scheduler's queue or these tasks needing only one execution? What about the priority / RM handling of a periodic initialization or fail recovery task? Further, I struggle to understand how useful RM could be in real RTOS implementations. If it is sufficient for real-world purposes, to what degree does task aperiodicity affect an RM scheduler's LUB and safety? Finally, I would like to see more about blocking in a RM context, as it was ignored by one of its main conditions in this work.

(e) If Apollo 11 had waited three and a half years for Liu and Layland to publish Rate Monotonic scheduling, I doubt they would have been much better off. While rate monotonic may have been a more effective and efficient scheduler than MIT had used prior (though that is uncertain and/or trivial), unexpected fast or aperiodic task requests would have overwhelmed the system regardless. In either case, the alarms would likely have gone off due to high load. The real savior was the restart protocol and failsafes which terminated low-importance tasks in favor of essential ones when the processor became overwhelmed.

3 yup did all that

4 RT-Clock

(a) The key function for the RT-Clock code is `delay_test()`. This function fetches the time from the clock, here the POSIX clock in real time (`CLOCK_REALTIME` as opposed to `CLOCK_MONOTONIC`, which logs time from system boot), with the `clock_gettime()` call. To get real time, we call it with the arguments `clock_gettime(CLOCK_REALTIME, *ts)`. `*ts` is a timespec, detailed in the manpage, which defines the time format (i.e. seconds, ms, us, ns). From what I can surmise, it seems that timespec should generally be in the clock's natural resolution (`clock_getres(clockid, &res)`).

(b) gettime() uses a vDSO syscall. I found this by listing all processes and finding the PID of the

`posix_clock` process with `ps -ef | grep posix_clock`. We can now search the /proc/pid/maps directory (/proc/ lets you access kernel data structures) to find info on what calls our process may have executed and filter with `cat proc/pid/maps | grep -E 'vdso|vsyscall'` the -E argument lets us input an expression for the search and only use grep once. However, this returns nothing! Or at least it did for me. Why? Well, we run another `ps -ef | grep posix_clock`. The pid was originally 9773 during execution - now it's 9780, and it increases every time. Okay, maybe it's because the code has already finished execution. No matter - you have a parent process ID (ppid), and you can use that for the /proc/ search instead. So after modifying that, we get the following output:

```
7f9ca41000-7f9ca42000 r-xp 00000000 00:00 0                              [vdso]
```

This means the only call we made from posix-clock was a vDSO call. vDSO stands for "virtual dynamic shared object", and vDSO syscalls are typically used for calls which care made often, especially bu the C library (per the vdso manpage). Unlike regular syscalls, which are slowed while pulling from kernel-space, vDSO calls are sped up by access to a shared library between kernel and user space.

(c) 1. Interrupt handler latency is the time between a service's interrupt request and the beginning of service execution. A low interrupt latency is favorable as it means the system is more responsive to IRQs and it allows for more flexibility in meeting deadlines.
2. Context switch time is the time it takes to switch from one task or thread to another. Context switching can meaningfully affect execution times as the operation requires caching the current thread state, retrieving the cached state of another task, and resuming execution. RTOS vendors seek to reduce this time as it can cut into task throughput, especially in high-load scenarios where threads may need to context switch frequently.
3. Low jitter/drift is extremely important for an RTOS. With high jitter or clock drift, deadlines are more difficult or impossible to guarantee – that's not a reality for RTOS vendors. The defining feature of an RTOS over other systems is the ability to guarantee deadlines and precisely specify worst-case errors. Without it, mission-critical systems would not have a very strong guarantee of system performance, even without external complications.

(d) The POSIX clock is rather precise, at least in a %-error sense. Every time it runs for 3 requested sleep seconds, it reports a value between about 3.00009 and 3.00010 seconds (between 90,000 and 100,000 ns). It is not very accurate, though, given that it has a resolution of 1 ns. The code does not specify a measure of target/threshold accuracy, just the requested time and the clock's resolution. However, my belief that the value the print statements report is the same as it truly took between `gettime()` calls is not strong, given that the next question asks us to time it.

(e) Over several runs, the time between two `gettime()` calls returning was approximately 150-200 nanoseconds. The call time is likely not the source of the overshoot.

(f) The error in sleep time has been reduced by half and now averages about 50,000 ns. The time of a `gettime()` call remains roughly the same. This may be because SCHED_FIFO (being first in, first out) is able to schedule the sleep time request task (in the do...while block) with a lower latency. This would lower the error in the final timestep, as it would enable more frequent checks of the sleep remaining by reducing wait between task executions.

5 Threading and pthreads

(a) The code in `/simplethread/` prints the thread idx and output of 12 sums $\sum_{i=0}^{n} i$ for n=11. The output is loosely in order of increasing i, though rows may be misplaced in the hierarchy by varying amounts; there doesn't seem to be a pattern of if or how it happens each run.
The code in `/rt_simplethread/` completes a similar task to the first, though it now uses real-time scheduling (SCHED_FIFO). However, it runs on a single thread and only outputs one iteration of the counter (for thread idx=0) before finishing. Execution time was consistently about 27 ms.
With `/rt_thread_improved/`, the code is implemented for all available threads of the system. For the raspberry pi, without hyperthreading, that's four cores giving you four threads total. Now, the summation task is done on each thread, and instead the task on each core is calculating $\sum_{i=1}^{n} 100i$ with $n = threadidx + 1$. So core 0, for instance, calculates the sum of each number up to 100. This is done concurrently, and while each thread's execution time differs slightly, they are always within 1 ms of each other. Execution time fell haphazardly between 0.5 and 1.5 seconds, with no real indication of what caused such variation. Other system or hidden higher-priority tasks could possibly have caused higher latency for some runs.

(b) Pthreads, seen in `incdecthread`, stands for POSIX threads, a language-agnostic standard for thread operations. Then, we see the two POSIX functions used are `pthread_create` and `pthread_join`. `pthread_create` creates a new thread and passes some function for it to complete. We can think of creating threads in pthreads in a more abstract sense as "forking" a thread, a parallel computing concept. If we consider the main() function calling `pthread_create()` as the initial thread completing some large task, we may delegate a subtask to be done in parallel and fork the current thread in two, resulting in our main thread and the newly forked subtask thread. Once the forked thread has completed its task, we need to then reunite it with the main thread using `pthread_join` in order to synchronize the tasks. This model is called the "fork-join model". The input arguments in `pthread_create(*threadref, *attributes, *start_routine, *params)` denote the intended location of the created thread, the attributes it's initialized with, the instructions it executes upon creation, and additional parameters, respectively. `pthread_join(thread, **return_val)` denote the thread to be joined and a pointer to a storage location for the thread's return value, if applicable. To make `incdecthread`... work, we have to somehow prioritize completing the incrementing thread first. Trying this by setting SCHED_FIFO priorities similar to the two real-time threads from the previous question, we get a function with slightly more predictable behavior, rather than inc and dec constantly fighting over `gsum`. However, it's not at all solved – there's plenty of mixing around the middle of execution rather than a smooth handoff from increasing to decreasing. To accomplish that, we use a semaphore to prevent decrements until incrementing is complete. We create the semaphore object `sem_inc` in the beginning definitions block and call `sem_init(&sem_inc, 0644, 0)`with 0644 indicating access permissions, in the body of main(). We then call `sem_post(&sem_inc)` at the end of `incThread` and we begin `decThread` with a `sem_wait(&sem_inc)` so it waits to execute until `incThread` is complete. At the end of main, we destroy the semaphore by calling `sem_destroy(&sem_inc)`. Et voila! Flawless deterministic execution, with `incThread` counting all the way up before handing it off to `decThread` to get down to 0.

(c) This schedule contains two services and an idle state. By implementing a SCHED_FIFO scheduler, for example, this could be done fairly simply with static priorities and essentially 2 services. The two services would have high priorities, with the task running 5x per LCM period having the highest and the second task just below that. Then, the idle state would have the lowest priority, though it doesn't really need to be implemented.

(d) I was hung up here for incredibly long because I cannot get the code to compile and work properly. I have coded a solution to the problem, but nonstandard libraries in C are something I can't quite

figure out how to get working. I installed the files, put them in my includepath, made sure I wasn't getting errors in my IDE, and it still didn't work. I have attached the code in the appendix. I used pthreads, syslog, and some libraries we've used before in this lab. I didn't know whether to make Sequencer its own thread, since we've tended to fork them from main(), but either way it should work.

6 Appendix

6.1 RT-Clock

6.1.1 Code

```
/****************************************************************************/
/* Function: nanosleep and POSIX 1003.1b RT clock demonstration             */
/*                                                                          */
/* Sam Siewert - 02/05/2011                                                 */
/*                                                                          */
/****************************************************************************/

#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <errno.h>

#define NSEC_PER_SEC (1000000000)
#define DELAY_TICKS (1)
#define ERROR (-1)
#define OK (0)

void end_delay_test(void);
static struct timespec sleep_time = {0, 0};
static struct timespec sleep_requested = {0, 0};
static struct timespec remaining_time = {0, 0};
static unsigned int sleep_count = 0;
pthread_t main_thread;
pthread_attr_t main_sched_attr;
int rt_max_prio, rt_min_prio, min;
struct sched_param main_param;

void print_scheduler(void)
{
   int schedType;

   schedType = sched_getscheduler(getpid());

   switch(schedType)
   {
     case SCHED_FIFO:
```

```
            printf("Pthread Policy is SCHED_FIFO\n");
            break;
      case SCHED_OTHER:
            printf("Pthread Policy is SCHED_OTHER\n");
        break;
      case SCHED_RR:
            printf("Pthread Policy is SCHED_OTHER\n");
            break;
      default:
        printf("Pthread Policy is UNKNOWN\n");
    }
}
int delta_t(struct timespec *stop, struct timespec *start, struct timespec *delta_t)
{
  int dt_sec=stop->tv_sec - start->tv_sec;
  int dt_nsec=stop->tv_nsec - start->tv_nsec;

  if(dt_sec >= 0)
  {
    if(dt_nsec >= 0)
    {
      delta_t->tv_sec=dt_sec;
      delta_t->tv_nsec=dt_nsec;
    }
    else
    {
      delta_t->tv_sec=dt_sec-1;
      delta_t->tv_nsec=NSEC_PER_SEC+dt_nsec;
    }
  }
  else
  {
    if(dt_nsec >= 0)
    {
      delta_t->tv_sec=dt_sec;
      delta_t->tv_nsec=dt_nsec;
    }
    else
    {
      delta_t->tv_sec=dt_sec-1;
      delta_t->tv_nsec=NSEC_PER_SEC+dt_nsec;
    }
  }
  return(OK);
}

static struct timespec rtclk_dt = {0, 0};
static struct timespec rtclk_start_time = {0, 0};
static struct timespec rtclk_stop_time = {0, 0};
static struct timespec rtclk_stop_time2 = {0, 0};
```

```
static struct timespec delay_error = {0, 0};
static struct timespec gettime_dt = {0, 0};

void *delay_test(void *threadID)
{
  int i;
  unsigned int max_sleep_calls=3;
  int flags = 0;
  struct timespec rtclk_resolution;

  sleep_count = 0;

  if(clock_getres(CLOCK_REALTIME, &rtclk_resolution) == ERROR)
  {
      perror("clock_getres");
      exit(-1);
  }
  else
  {
      printf("\n\nPOSIX Clock demo using system RT clock with resolution:\n\t%ld secs, %ld micros
  }

  /* run test for 3 seconds */
  sleep_time.tv_sec=3;
  sleep_time.tv_nsec=0;
  sleep_requested.tv_sec=sleep_time.tv_sec;
  sleep_requested.tv_nsec=sleep_time.tv_nsec;

  /* start time stamp */
  clock_gettime(CLOCK_REALTIME, &rtclk_start_time);

  /* request sleep time and repeat if time remains */
  do
  {
      nanosleep(&sleep_time, &remaining_time);

      sleep_time.tv_sec = remaining_time.tv_sec;
      sleep_time.tv_nsec = remaining_time.tv_nsec;
      sleep_count++;
  }
  while (((remaining_time.tv_sec > 0) || (remaining_time.tv_nsec > 0)) && (sleep_count < max_slee

  clock_gettime(CLOCK_REALTIME, &rtclk_stop_time);
  clock_gettime(CLOCK_REALTIME, &rtclk_stop_time2);

  delta_t(&rtclk_stop_time, &rtclk_start_time, &rtclk_dt);
  delta_t(&rtclk_dt, &sleep_requested, &delay_error);
  delta_t(&rtclk_stop_time2, &rtclk_stop_time, &gettime_dt);
  end_delay_test();
```

```
}

void end_delay_test(void)
{
  printf("\n");
  printf("RT clock start seconds = %ld, nanoseconds = %ld\n",
         rtclk_start_time.tv_sec, rtclk_start_time.tv_nsec);

  printf("RT clock stop seconds = %ld, nanoseconds = %ld\n",
         rtclk_stop_time.tv_sec, rtclk_stop_time.tv_nsec);

  printf("RT clock DT seconds = %ld, nanoseconds = %ld\n",
         rtclk_dt.tv_sec, rtclk_dt.tv_nsec);

  printf("Requested sleep seconds = %ld, nanoseconds = %ld\n",
         sleep_requested.tv_sec, sleep_requested.tv_nsec);

  printf("\n");
  printf("Sleep loop count = %ld\n", sleep_count);
  printf("RT clock delay error = %ld, nanoseconds = %ld\n",
         delay_error.tv_sec, delay_error.tv_nsec);
  printf("gettime() call time delta = %ld, nanoseconds = %ld\n", gettime_dt.tv_sec, gettime_dt.tv

  exit(0);

}

#define RUN_RT_THREAD
void main(void)
{
   int rc, scope;

   printf("Before adjustments to scheduling policy:\n");
   print_scheduler();

#ifdef RUN_RT_THREAD
   pthread_attr_init(&main_sched_attr);
   pthread_attr_setinheritsched(&main_sched_attr, PTHREAD_EXPLICIT_SCHED);
   pthread_attr_setschedpolicy(&main_sched_attr, SCHED_FIFO);
   rt_max_prio = sched_get_priority_max(SCHED_FIFO);
   rt_min_prio = sched_get_priority_min(SCHED_FIFO);
   main_param.sched_priority = rt_max_prio;
   rc=sched_setscheduler(getpid(), SCHED_FIFO, &main_param);
   if (rc)
   {
       printf("ERROR; sched_setscheduler rc is %d\n", rc);
       perror("sched_setschduler"); exit(-1);
   }
   printf("After adjustments to scheduling policy:\n");
   print_scheduler();
```

```
   main_param.sched_priority = rt_max_prio;
   pthread_attr_setschedparam(&main_sched_attr, &main_param);
   rc = pthread_create(&main_thread, &main_sched_attr, delay_test, (void *)0);
   if (rc)
   {
       printf("ERROR; pthread_create() rc is %d\n", rc);
       perror("pthread_create");
       exit(-1);
   }
   pthread_join(main_thread, NULL);
   if(pthread_attr_destroy(&main_sched_attr) != 0)
     perror("attr destroy");
#else
   delay_test((void *)0);
#endif
   printf("TEST COMPLETE\n");
}
```

6.1.2 Figures

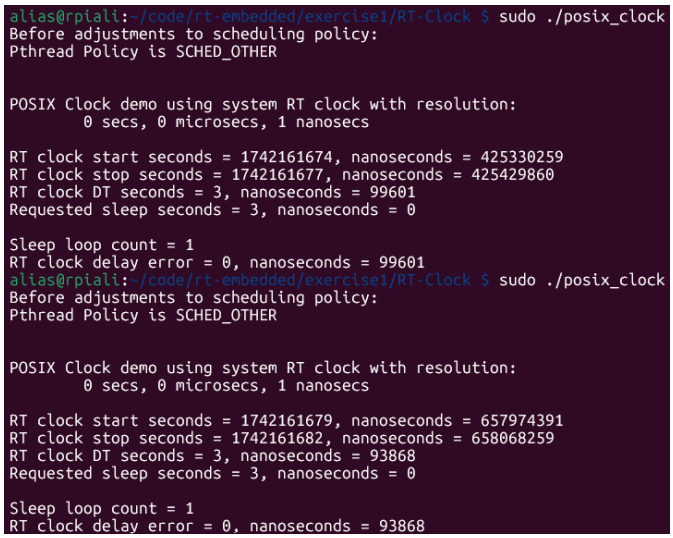

Figure 3: The posix_clock code running nominally.

6.2 Pthreads

6.2.1 Code: incdecthread/pthread.c

```
#define _GNU_SOURCE

#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <sched.h>
#include <semaphore.h>
```

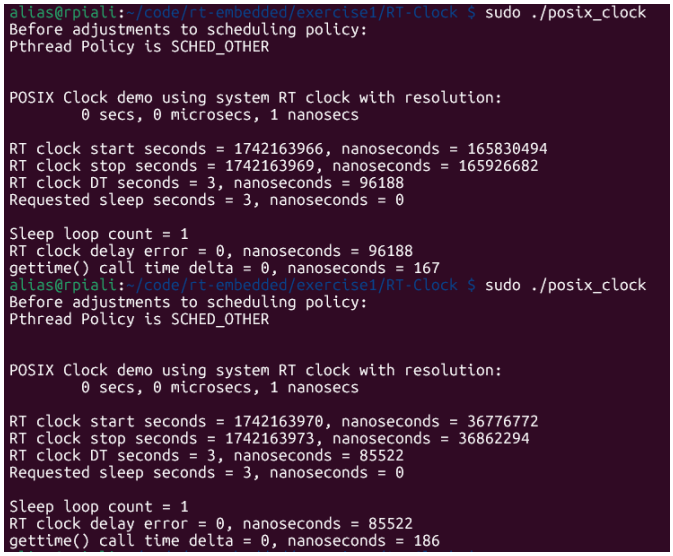

Figure 4: Modified posix_clock code output showing the time it takes to call gettime().

```
#include <unistd.h>

#define COUNT  1000

typedef struct
{
    int threadIdx;
} threadParams_t;

// POSIX thread declarations and scheduling attributes
//
pthread_t threads[2];
threadParams_t threadParams[2];
pthread_attr_t rt_sched_attr[2];
int rt_max_prio, rt_min_prio;
struct sched_param rt_param[2];
struct sched_param main_param;
pid_t mainpid;
sem_t sem_inc;
// Unsafe global
int gsum=0;

void *incThread(void *threadp)
{
    int i;
    threadParams_t *threadParams = (threadParams_t *)threadp;

    for(i=0; i<COUNT; i++)
    {
        gsum=gsum+i;
        printf("Increment thread idx=%d, gsum=%d\n", threadParams->threadIdx, gsum);
```

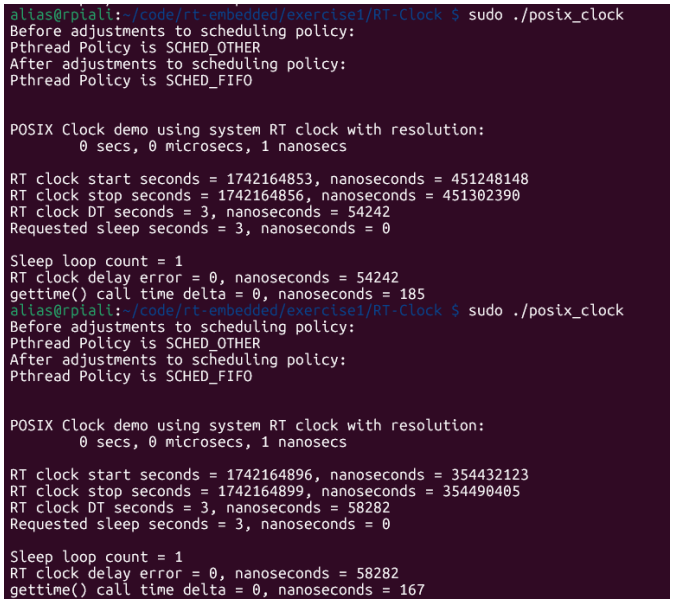

Figure 5: posix_clock running with SCHED_FIFO

```
    }
    sem_post(&sem_inc);
}
void *decThread(void *threadp)
{
    int i;
    threadParams_t *threadParams = (threadParams_t *)threadp;
    sem_wait(&sem_inc);
    for(i=0; i<COUNT; i++)
    {
        gsum=gsum-i;
        printf("Decrement thread idx=%d, gsum=%d\n", threadParams->threadIdx, gsum);
    }
}

int main (int argc, char *argv[])
{
   int rc;
   int i=0;

   mainpid = getpid();
   cpu_set_t cpuset;
   //CPU_ZERO(&cpuset);
   // without this it becomes very upset?
   //CPU_SET(0, &cpuset);

    // get mix, max prio. since I'm only using FIFO I could really just have these
    // values be 30 and 29 or whatever, but I'll code like this might ever be used again.
   rt_max_prio = sched_get_priority_max(SCHED_FIFO);
   rt_min_prio = sched_get_priority_min(SCHED_FIFO);
```

```
   rc=sched_getparam(mainpid, &main_param);
   main_param.sched_priority = rt_max_prio;
   rc=sched_setscheduler(getpid(), SCHED_FIFO, &main_param);

   sem_init(&sem_inc, 0644, 0);

   rc=pthread_attr_init(&rt_sched_attr[i]);
   rc=pthread_attr_setinheritsched(&rt_sched_attr[i], PTHREAD_EXPLICIT_SCHED);
   rc=pthread_attr_setschedpolicy(&rt_sched_attr[i], SCHED_FIFO);
   rc=pthread_attr_setaffinity_np(&rt_sched_attr[i], sizeof(cpu_set_t), &cpuset);
    rt_param[i].sched_priority=rt_max_prio-i-1;
    pthread_attr_setschedparam(&rt_sched_attr[i], &rt_param[i]);

   threadParams[i].threadIdx=i;
   pthread_create(&threads[i],   // pointer to thread descriptor
                  &rt_sched_attr[i],     // use default attributes
                  incThread, // thread function entry point
                  (void *)&(threadParams[i]) // parameters to pass in
                 );

   i++;

   rc=pthread_attr_init(&rt_sched_attr[i]);
   rc=pthread_attr_setinheritsched(&rt_sched_attr[i], PTHREAD_EXPLICIT_SCHED);
   rc=pthread_attr_setschedpolicy(&rt_sched_attr[i], SCHED_FIFO);
   rc=pthread_attr_setaffinity_np(&rt_sched_attr[i], sizeof(cpu_set_t), &cpuset);
   rt_param[i].sched_priority=rt_max_prio-i-1;
   pthread_attr_setschedparam(&rt_sched_attr[i], &rt_param[i]);
   threadParams[i].threadIdx=i;
   pthread_create(&threads[i], &rt_sched_attr[i], decThread, (void *)&(threadParams[i]));
   for(i=0; i<2; i++)
     pthread_join(threads[i], NULL);
   printf("TEST COMPLETE\n");
   sem_destroy(&sem_inc);
}
```

6.2.2 Code: lab1.c

```
/****************************************************************************/
/*                                                                          */
/* Sam Siewert - 2005                                                       */
/*                                                                          */
/****************************************************************************/
#define _GNU_SOURCE
#include "vxWorks.h"
#include "semLib.h"
#include "sysLib.h"
#include "wvLib.h"
#include <syslog.h>
```

```
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>

#define FIB_LIMIT_FOR_32_BIT 47
#define FIB10_ITERATIONS 170000  //meant to be ~10 ms
#define FIB20_ITERATIONS 340000

SEM_ID semF10, semF20;
int abortTest = 0;
UINT32 seqIterations = FIB_LIMIT_FOR_32_BIT;
UINT32 idx = 0, jdx = 1;
UINT32 fib = 0, fib0 = 0, fib1 = 1;
UINT32 fib10Cnt=0, fib20Cnt=0;
char ciMarker[]="CI";


#define FIB_TEST(seqCnt, iterCnt)      \
   for(idx=0; idx < iterCnt; idx++)    \
   {                                   \
      fib = fib0 + fib1;               \
      while(jdx < seqCnt)              \
      {                                \
         fib0 = fib1;                  \
         fib1 = fib;                   \
         fib = fib0 + fib1;            \
         jdx++;                        \
      }                                \
   }                                   \


/* Iterations, 2nd arg must be tuned for any given target type
   using windview

   170000 <= 10 msecs on Pentium at home

   Be very careful of WCET overloading CPU during first period of
   LCM.

 */

// get current time in milliseconds (using CLOCK_MONOTONIC)
double get_time_ms() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec * 1000.0 + ts.tv_nsec / 1.0e6;
}
```

```
void fib10(void)
{
   while(!abortTest)
   {
    semTake(semF10, WAIT_FOREVER);
    double start = get_time_ms();
    FIB_TEST(seqIterations, FIB10_ITERATIONS);
    fib10Cnt++;
    double end = get_time_ms();
    double elapsed = end - start;
    printf("Fib10 executed: %f ms", elapsed);
   }
}

void fib20(void)
{
   while(!abortTest)
   {
    semTake(semF20, WAIT_FOREVER);
    double start = get_time_ms();
    FIB_TEST(seqIterations, FIB20_ITERATIONS);
    fib20Cnt++;
    double end = get_time_ms();
    double elapsed = end - start;
    printf("Fib10 executed: %f ms", elapsed);
   }
}

void shutdown(void)
{
 abortTest=1;
  pthread_join(thd_fib10, NULL);
  pthread_join(thd_fib20, NULL);
  pthread_join(thd_seq, NULL);
  sem_destroy(&semF10);
  sem_destroy(&semF20);
  syslog(LOG_INFO, "RT Schedule Program ending");
  closelog();
  return 0;
}

void Sequencer(void)
{

  printf("Starting Sequencer\n");

  /* Just to be sure we have 1 msec tick and TOs */
  sysClkRateSet(1000);

  /* Set up service release semaphores */
```

```
  semF10 = semBCreate(SEM_Q_FIFO, SEM_EMPTY);
  semF20 = semBCreate(SEM_Q_FIFO, SEM_EMPTY);

  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
  pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
  struct sched_param param;
  param.sched_priority = 80;
  pthread_attr_setschedparam(&attr, &param);

/*
  if(taskSpawn("serviceF10", 21, 0, 8192, fib10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) == ERROR)
  {
    printf("F10 task spawn failed\n");
  }
  else
    printf("F10 task spawned\n");

  if(taskSpawn("serviceF20", 22, 0, 8192, fib20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) == ERROR)
  {
    printf("F20 task spawn failed\n");
  }
  else
    printf("F20 task spawned\n");

*/

  pthread_t thd_fib10, thd_fib20;
  if (pthread_create(&thd_fib10, &attr, fib10, NULL) != 0) {
      printf("F10 thread spawn failed\n");
      exit(EXIT_FAILURE);
  }
  if (pthread_create(&thd_fib20, &attr, fib20, NULL) != 0) {
      printf("F20 thread spawn failed\n");
      exit(EXIT_FAILURE);
  }

  /* Simulate the C.I. for S1 and S2 and mark on windview and log
     wvEvent first because F10 and F20 can preempt this task!
   */
  if(wvEvent(0xC, ciMarker, sizeof(ciMarker)) == ERROR)
   printf("WV EVENT ERROR\n");
  semGive(semF10); semGive(semF20);


  /* Sequencing loop for LCM phasing of S1, S2
   */
  while(!abortTest)
  {
```

```
   /* Basic sequence of releases after CI */
    syslog(LOG_INFO, "CI: releasing Fib10 and Fib20");
    taskDelay(20); semGive(semF10);
    syslog(LOG_INFO, "Sequencer: released Fib10 at +20ms");
    taskDelay(20); semGive(semF10);
    syslog(LOG_INFO, "Sequencer: released Fib10 at +40ms");
    taskDelay(10); semGive(semF20);
    syslog(LOG_INFO, "Sequencer: released Fib20 at +50ms");
    taskDelay(10); semGive(semF10);
    syslog(LOG_INFO, "Sequencer: released Fib10 at +60ms");
    taskDelay(20); semGive(semF10);
    syslog(LOG_INFO, "Sequencer: released Fib10 at +80ms");
    taskDelay(20);
    syslog(LOG_INFO, "Sequencer: Done at +100ms");

   /* back to C.I. conditions, log event first due to preemption */
   if(wvEvent(0xC, ciMarker, sizeof(ciMarker)) == ERROR)
    printf("WV EVENT ERROR\n");
   semGive(semF10); semGive(semF20);
  }

  pthread_exit(NULL);
}

void start(void)
{
 abortTest=0;
  pthread_t thd_seq;
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
  pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
  struct sched_param param;
  param.sched_priority = 80;
  pthread_attr_setschedparam(&attr, &param);

  if(pthread_create(&thd_seq, &attr, Sequencer, NULL) != 0) {
      printf("Sequencer thread spawn failed.");
      exit(EXIT_FAILURE);
  }
}
```

6.2.3 Figures

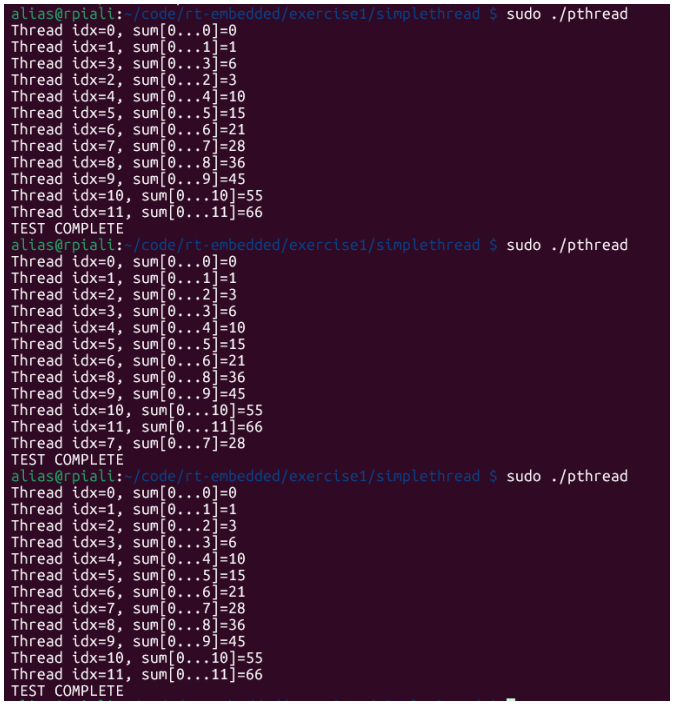

Figure 6: The simplethread/pthread.c code running nominally.

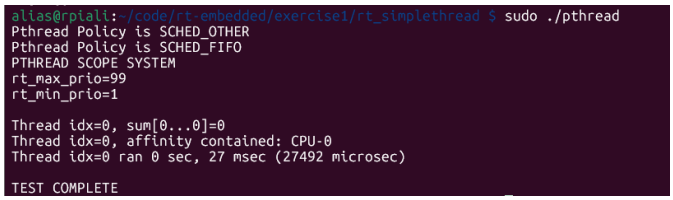

Figure 7: The realtime simple thread code runnning nominally.

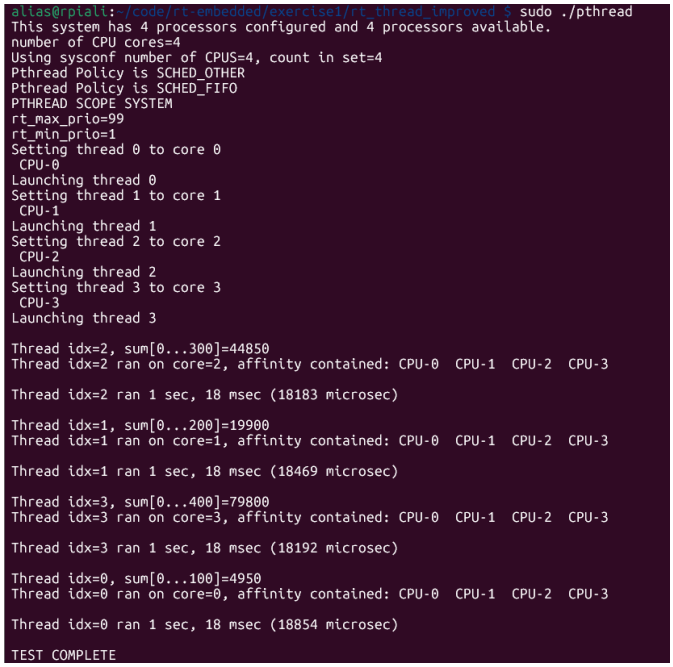

Figure 8: The proper realtime thread code running properly nominally.

Figure 9: Output of the code from incdecthread/ centered around the middle of the sequence. Notice that the increment and decrement do not smoothly hand off, but interlace together where they shouldn't. However, initial and final stretches of this output show expected behavior (increment -¿ decrement).

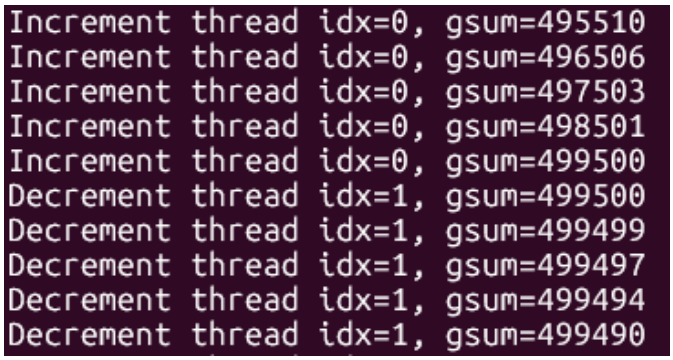

Figure 10: The output of the incdecthread/ code using semaphores and FIFO, right at the semaphore changing hands. Here, increment fully completes its task before handing off to decrement.