Lab 2: RTOS Kernel

Xinyuan (Allen) Pan & Qinyu (Jack) Diao

Objective

The objective of this lab was to develop the RTOS kernel. We coordinated multiple foreground and background threads using a round robin scheduler and spinlock cooperative semaphores for thread synchronization. The threads communicated with each other through FIFOs and a mailbox. Switch interrupts, thread sleep and other functionality were also implemented.

Software Design

```
enum State {
                FREE,
                ACTIVE,
                SLEEP,
                BLOCKED
};

typedef struct Sema4{
  long value;   // > 0 means free, otherwise means busy
// add other components here, if necessary to implement blocking
} Sema4Type;


/*
 *        Thread Control Block structure
 */
typedef struct tcb {
        int32_t *sp;        // ** MUST be the first field ** saved stack pointer (not used by active thread)
        struct tcb *next;   // ** MUST be the second field
        struct tcb *prev;
        enum State state;
        int id;
        uint32_t sleepTimeLeft;    // number of cycles left the thread needs to remain in sleep state
        Sema4Type *blocked;
//      uint32_t priority;
} tcbType;


extern tcbType *RunPt;


// ******** OS_Init ************
// initialize operating system, disable interrupts until OS_Launch
// initialize OS controlled I/O: serial, ADC, systick, LaunchPad I/O and timers
// input:  none
// output: none
void OS_Init(void);

// ******** OS_InitSemaphore ************
// initialize semaphore
// input:  pointer to a semaphore
// output: none
void OS_InitSemaphore(Sema4Type *semaPt, long value);

// ******** OS_Wait ************
// decrement semaphore
// Lab2 spinlock
// Lab3 block if less than zero
// input:  pointer to a counting semaphore
// output: none
void OS_Wait(Sema4Type *semaPt);

// ******** OS_Signal ************
// increment semaphore
// Lab2 spinlock
// Lab3 wakeup blocked thread if appropriate
// input:  pointer to a counting semaphore
// output: none
void OS_Signal(Sema4Type *semaPt);

// ******** OS_bWait ************
// Lab2 spinlock, set to 0
// Lab3 block if less than zero
// input:  pointer to a binary semaphore
// output: none
void OS_bWait(Sema4Type *semaPt);

// ******** OS_bSignal ************
// Lab2 spinlock, set to 1
// Lab3 wakeup blocked thread if appropriate
// input:  pointer to a binary semaphore
// output: none
void OS_bSignal(Sema4Type *semaPt);

//******** OS_AddThread ***************
// add a foregound thread to the scheduler
// Inputs: pointer to a void/void foreground task
//         number of bytes allocated for its stack
//         priority, 0 is highest, 5 is the lowest
// Outputs: 1 if successful, 0 if this thread can not be added
// stack size must be divisable by 8 (aligned to double word boundary)
// In Lab 2, you can ignore both the stackSize and priority fields
// In Lab 3, you can ignore the stackSize fields
```

```
int OS_AddThread(void(*task)(void),
   unsigned long stackSize, unsigned long priority);

//******** OS_Id ***************
// returns the thread ID for the currently running thread
// Inputs: none
// Outputs: Thread ID, number greater than zero
unsigned long OS_Id(void);

//******** OS_AddPeriodicThread ***************
// add a background periodic task
// typically this function receives the highest priority
// Inputs: pointer to a void/void background function
//         period given in system time units (12.5ns)
//         priority 0 is the highest, 5 is the lowest
// Outputs: 1 if successful, 0 if this thread can not be added
// You are free to select the time resolution for this function
// It is assumed that the user task will run to completion and return
// This task can not spin, block, loop, sleep, or kill
// This task can call OS_Signal  OS_bSignal   OS_AddThread
// This task does not have a Thread ID
// In lab 2, this command will be called 0 or 1 times
// In lab 2, the priority field can be ignored
// In lab 3, this command will be called 0 1 or 2 times
// In lab 3, there will be up to four background threads, and this priority field
//          determines the relative priority of these four threads
int OS_AddPeriodicThread(void(*task)(void),
   uint32_t period, uint32_t priority);

//******** OS_AddSW1Task ***************
// add a background task to run whenever the SW1 (PF4) button is pushed
// Inputs: pointer to a void/void background function
//         priority 0 is the highest, 5 is the lowest
// Outputs: 1 if successful, 0 if this thread can not be added
// It is assumed that the user task will run to completion and return
// This task can not spin, block, loop, sleep, or kill
// This task can call OS_Signal  OS_bSignal   OS_AddThread
// This task does not have a Thread ID
// In labs 2 and 3, this command will be called 0 or 1 times
// In lab 2, the priority field can be ignored
// In lab 3, there will be up to four background threads, and this priority field
//          determines the relative priority of these four threads
int OS_AddSW1Task(void(*task)(void), unsigned long priority);

//******** OS_AddSW2Task ***************
// add a background task to run whenever the SW2 (PF0) button is pushed
// Inputs: pointer to a void/void background function
//         priority 0 is highest, 5 is lowest
// Outputs: 1 if successful, 0 if this thread can not be added
// It is assumed user task will run to completion and return
// This task can not spin block loop sleep or kill
// This task can call issue OS_Signal, it can call OS_AddThread
// This task does not have a Thread ID
// In lab 2, this function can be ignored
// In lab 3, this command will be called will be called 0 or 1 times
// In lab 3, there will be up to four background threads, and this priority field
//          determines the relative priority of these four threads
int OS_AddSW2Task(void(*task)(void), unsigned long priority);

// ******** OS_Sleep ************
// place this thread into a dormant state
// input:  number of msec to sleep
// output: none
// You are free to select the time resolution for this function
// OS_Sleep(0) implements cooperative multitasking
void OS_Sleep(unsigned long sleepTime);

// ******** OS_Kill ************
// kill the currently running thread, release its TCB and stack
// input:  none
// output: none
void OS_Kill(void);

// ******** OS_Suspend ************
// suspend execution of currently running thread
// scheduler will choose another thread to execute
// Can be used to implement cooperative multitasking
// Same function as OS_Sleep(0)
// input:  none
// output: none
void OS_Suspend(void);

// ******** OS_Fifo_Init ************
// Initialize the Fifo to be empty
// Inputs: size
// Outputs: none
// In Lab 2, you can ignore the size field
// In Lab 3, you should implement the user-defined fifo size
// In Lab 3, you can put whatever restrictions you want on size
//    e.g., 4 to 64 elements
//    e.g., must be a power of 2,4,8,16,32,64,128
void OS_Fifo_Init(unsigned long size);

// ******** OS_Fifo_Put ************
// Enter one data sample into the Fifo
// Called from the background, so no waiting
// Inputs:  data
// Outputs: true if data is properly saved,
//          false if data not saved, because it was full
// Since this is called by interrupt handlers
//  this function can not disable or enable interrupts
int OS_Fifo_Put(unsigned long data);

// ******** OS_Fifo_Get ************
// Remove one data sample from the Fifo
// Called in foreground, will spin/block if empty
// Inputs:  none
// Outputs: data
unsigned long OS_Fifo_Get(void);

// ******** OS_Fifo_Size ************
// Check the status of the Fifo
// Inputs: none
// Outputs: returns the number of elements in the Fifo
//          greater than zero if a call to OS_Fifo_Get will return right away
//          zero or less than zero if the Fifo is empty
//          zero or less than zero if a call to OS_Fifo_Get will spin or block
long OS_Fifo_Size(void);

// ******** OS_MailBox_Init ************
// Initialize communication channel
// Inputs:  none
// Outputs: none
void OS_MailBox_Init(void);

// ******** OS_MailBox_Send ************
// enter mail into the MailBox
// Inputs:  data to be sent
// Outputs: none
// This function will be called from a foreground thread
// It will spin/block if the MailBox contains data not yet received
void OS_MailBox_Send(unsigned long data);

// ******** OS_MailBox_Recv ************
// remove mail from the MailBox
// Inputs:  none
// Outputs: data received
// This function will be called from a foreground thread
// It will spin/block if the MailBox is empty
unsigned long OS_MailBox_Recv(void);

// ******** OS_Time ************
// return the system time
// Inputs:  none
// Outputs: time in 12.5ns units, 0 to 4294967295
// The time resolution should be less than or equal to 1us, and the precision 32 bits
// It is ok to change the resolution and precision of this function as long as
//   this function and OS_TimeDifference have the same resolution and precision
unsigned long OS_Time(void);
```

```
// ******** OS_TimeDifference ************
// Calculates difference between two times
// Inputs:  two times measured with OS_Time
// Outputs: time difference in 12.5ns units
// The time resolution should be less than or equal to 1us, and the precision at least 12 bits
// It is ok to change the resolution and precision of this function as long as
//   this function and OS_Time have the same resolution and precision
unsigned long OS_TimeDifference(unsigned long start, unsigned long stop);

// ******** OS_ClearMsTime ************
// sets the system time to zero (from Lab 1)
// Inputs:  none
// Outputs: none
// You are free to change how this works
void OS_ClearMsTime(void);

// ******** OS_MsTime ************
// reads the current time in msec (from Lab 1)
// Inputs:  none
// Outputs: time in ms units
// You are free to select the time resolution for this function
// It is ok to make the resolution to match the first call to OS_AddPeriodicThread
unsigned long OS_MsTime(void);

//******** OS_Launch ***************
// start the scheduler, enable interrupts
// Inputs: number of 12.5ns clock cycles for each time slice
//         you may select the units of this parameter
// Outputs: none (does not return)
// In Lab 2, you can ignore the theTimeSlice field
// In Lab 3, you should implement the user-defined TimeSlice field
// It is ok to limit the range of theTimeSlice to match the 24-bit SysTick
void OS_Launch(unsigned long theTimeSlice);
```

Measurement Data

1) Plots

Figure 2.1: Cooperative

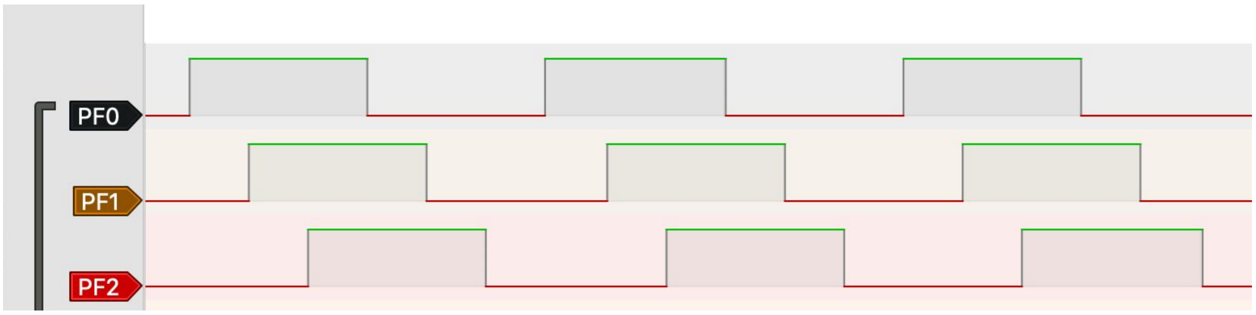


Figure 2.2: Preemptive (F1 is SysTick)

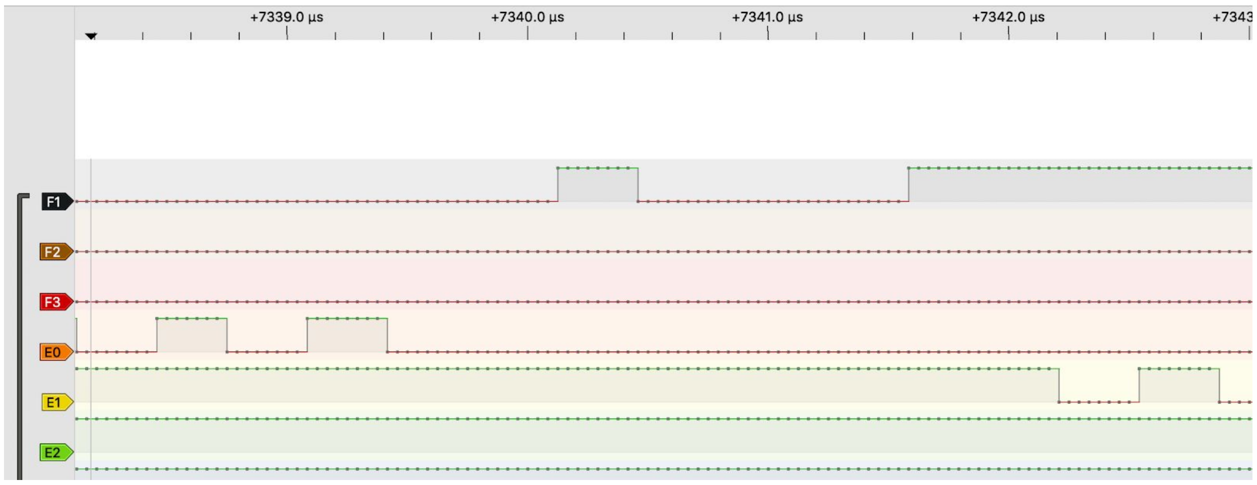


Figure 2.3: Preemptive (Zoomed out)

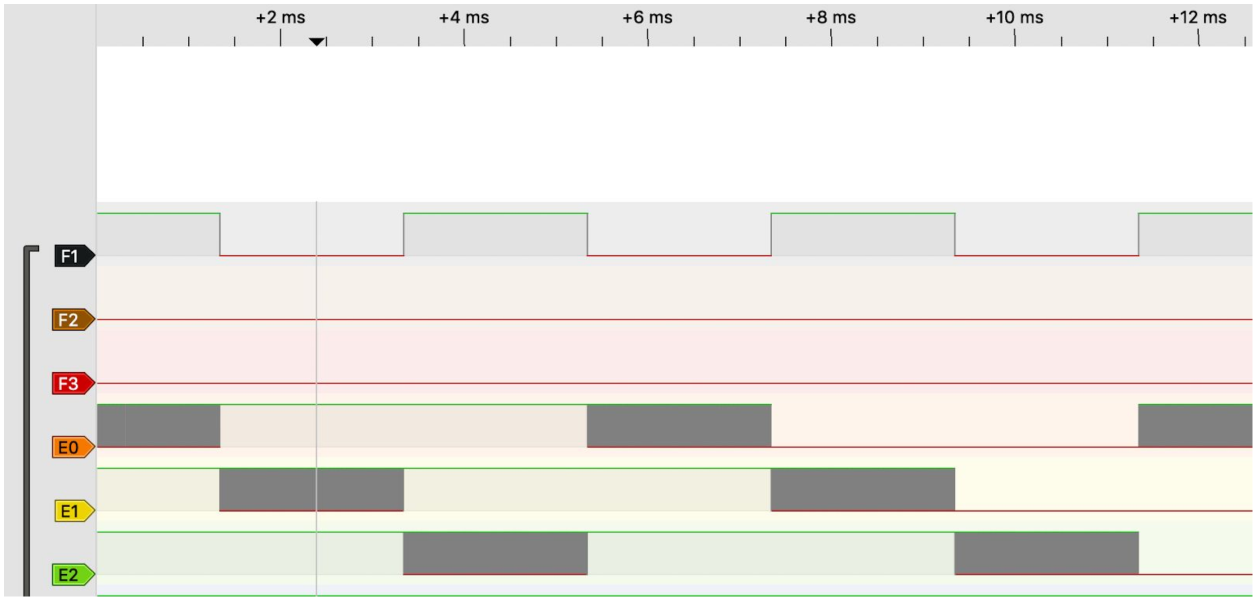
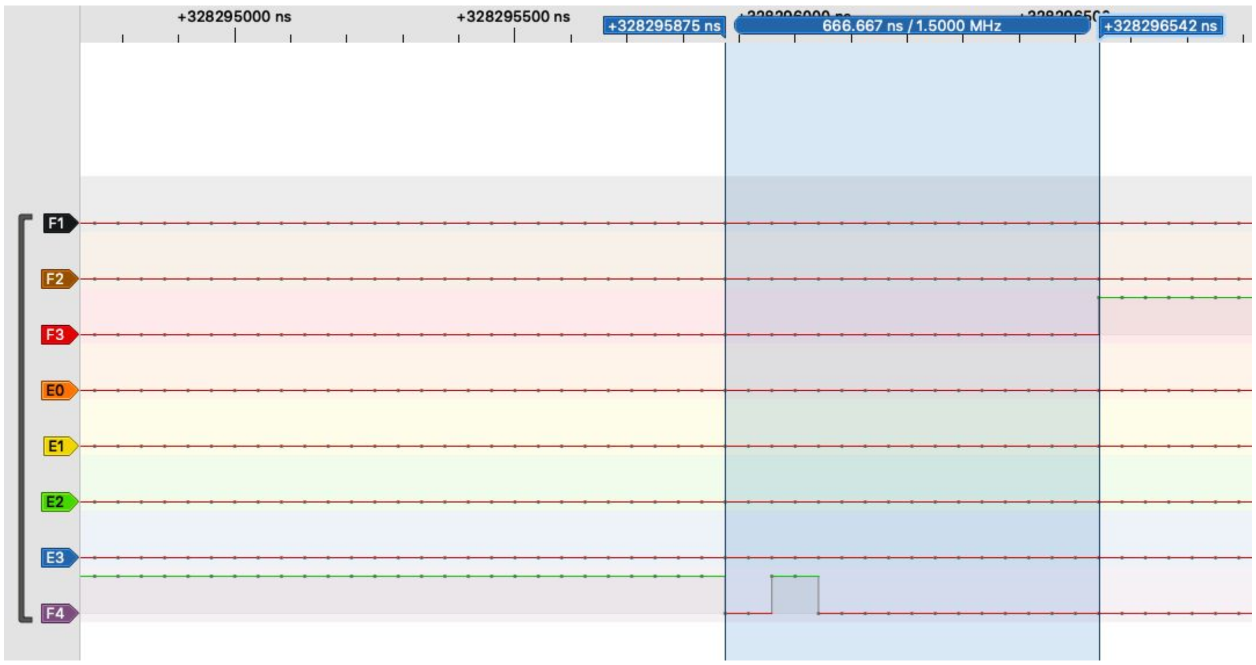


Figure 2.4: Latency of switch triggered thread switch (PF3 is switch ISR, F4 is switch)



Time spent in switch ISR

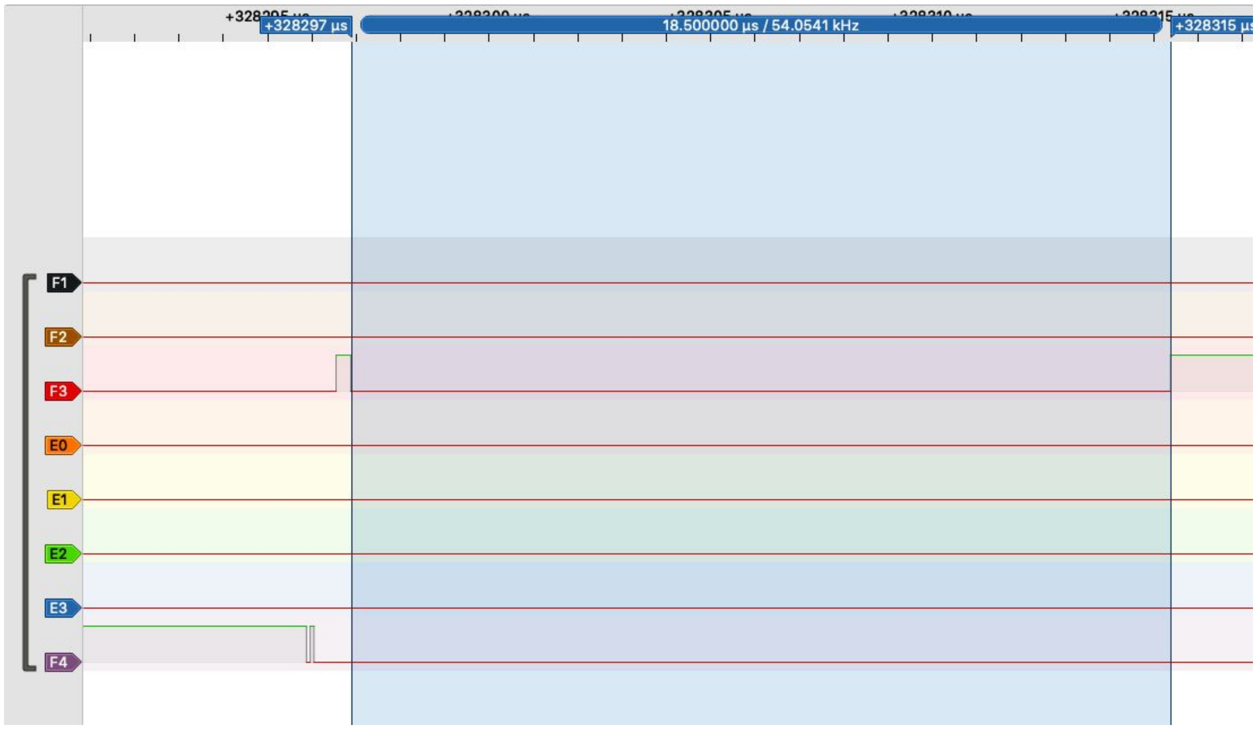
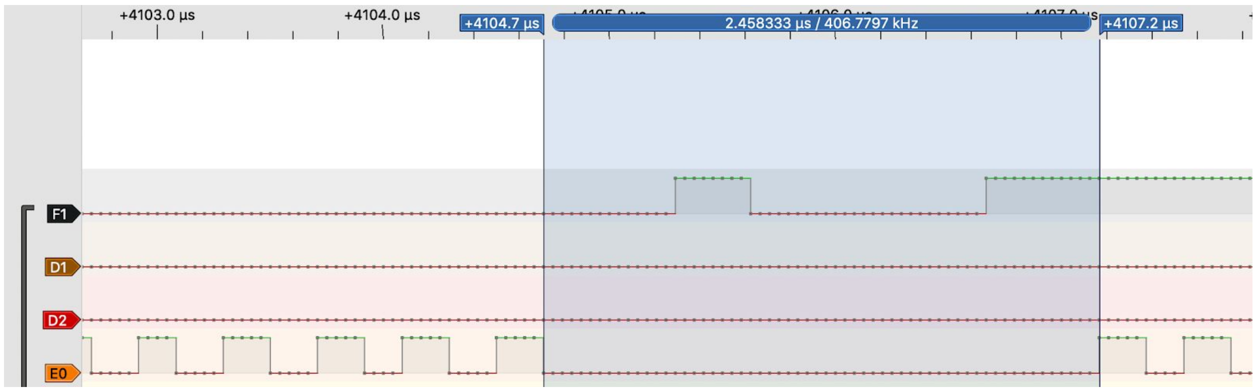


Figure 2.8: Thread-switch time

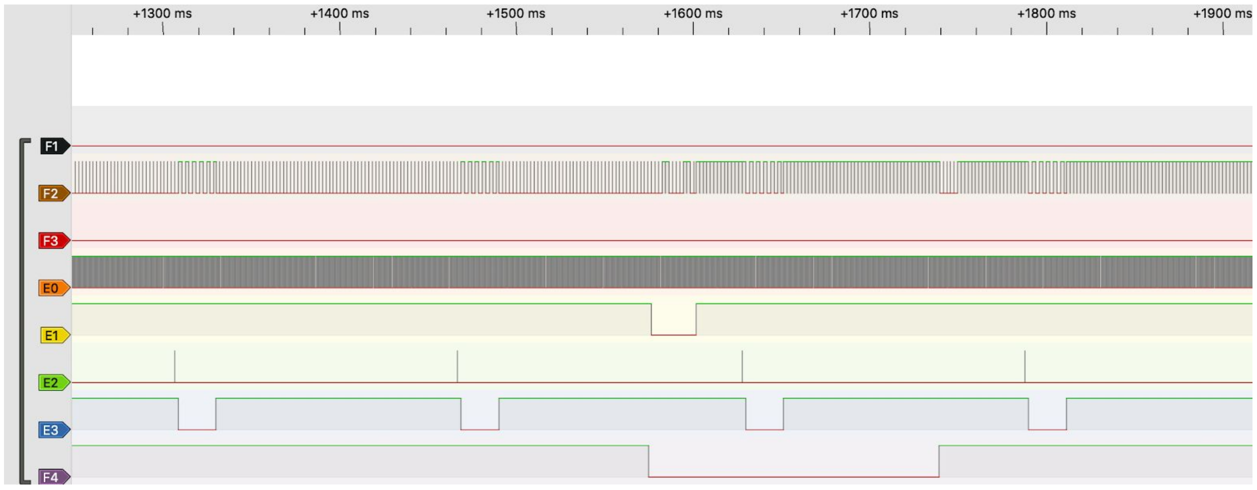


2) Measurement of the thread-switch time

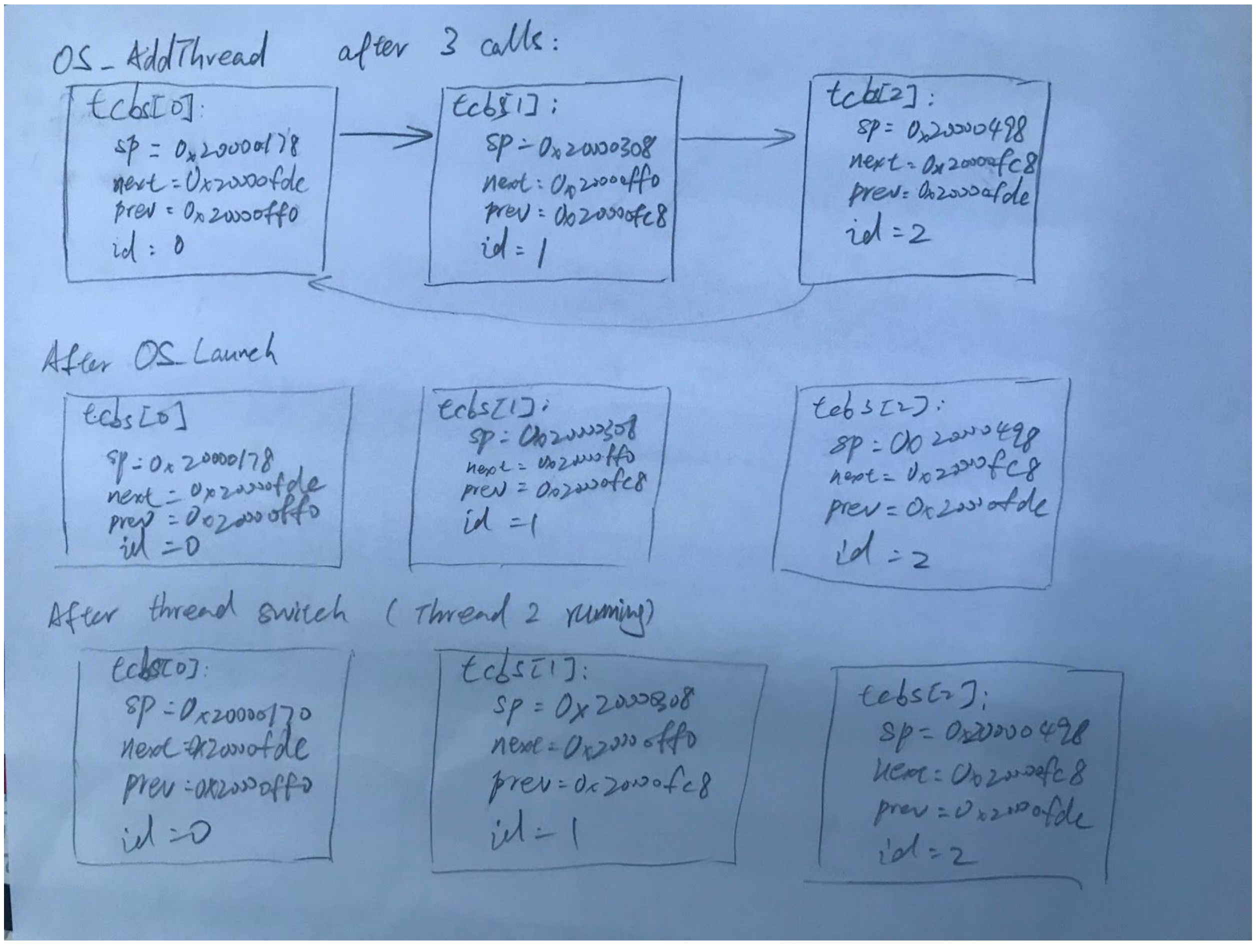
As shown in figure 2.8, the thread-switch time is 2.45 µs, including the LED-toggle time.

3) Profiling Plot

E0 profiles DAS main thread
E1 profiles ButtonWork thread created by switch 1 touch
E2 profiles Consumer main thread
E3 profiles Display main thread
F4 is the switch, negative logic
F2 is toggled everytime thread switch occurs



4) Sketches



5) Table showing performance measurements versus sizes of OS_FIFO_SIZE and timeslices

FIFOSize	TIMESLICE (ms)	DataLost	Jitter (μs)	PIDWork
2	2	4	1.3	10372
4	2	0	1.3	10425
16	1	0	1.3	10381
16	2	0	1.3	10425
16	10	0	1.2	10460

6) Table showing performance measurements with and without debugging instruments

Debug Profiling	DataLost	Jitter (µs)	PIDWork
ON	0	1.3	10425
OFF	0	1.2	10572

Analysis and Discussion

1) Why did the time jitter in my solution jump from 4 to 6 µs when interpreter I/O occurred?

When dealing with serial I/O, there are likely a lot of critical sections, where we disable the interrupt. Therefore, the interrupt request from DAS background thread may not be served immediately and cause jitter.

2) Justify why Task 3 has no time jitter on its ADC sampling.

1. Producer has a lower priority than DAS, so DAS interrupt can be served right away
2. Interrupt is never disabled in Consumer or Display. The communication between those threads are through FIFO or Mailbox, which are utilizing semaphore. Even if wait is spinning, it allows interrupts.

3) There are four (or more) interrupts in this system DAS, ADC, Select, and SysTick (thread switch). Justify your choice of hardware priorities in the NVIC?

DAS:	priority 1
ADC:	priority 2
Select:	priority 2
SysTick:	priority 7

SysTick handler must have a priority of 7, since it is not supposed to interrupt any background threads.
DAS is configured to have the highest priority in order to minimize the jitter.
ADC and Select's priorities need to be higher than SysTick, and lower than DAS. It may be better if ADC has higher priority than Select, since it is more real-time (even though there is no penalty for not exactly sampling at the given frequency in this lab.)

4) Explain what happens if your stack size is too small. How could you detect stack overflow? How could you prevent stack overflow from crashing the OS?

If stack size is too small, stack overflow is more likely to happen. That means there are too many local variables being allocated for a given thread and the stack is not big enough to contain, so they are being written to an area that does not belong to the stack.

If we know beforehand that how big the stack is allocated for each thread (which is defined by the function pointed to by the initial stack pointer), then we can write an magic number to the end of the stack. If the magic number is overwritten, that means stack is overflowed.
In the particular case of this lab, where we do not explicitly allocate a stack ourselves, we can only hope!

As soon as we detect the stack overflow, we can kill that thread and return an error code, which should be catched and properly handled by the application program. As long as we kill the thread, the OS won't crush, but the application may not run correctly.

5) Both Consumer and Display have an OS_Kill() at the end. Do these OS_Kills always execute, sometime execute, or never execute? Explain.

In every iteration, Consumer collects 64 samples from FIFO, then NumSamples is compared to RUNLENGTH. Since NumSamples is incremented in Producer no matter the data is lost or not, it may happen such that NumSamples is greater than the total sample being put in FIFO. In such a situation, Consumer may not be able to collect the 64 samples in the final iteration. So even though NumSamples == RUNLENGTH, Consumer cannot get out from Wait of the semaphore. Therefore, in Consumer OS_Kill sometimes executes.

Similarly, since for every MailBox_Recv to be executed, MailBox_Send needs to execute first. So in Display OS_Kill sometimes executes.

6) The interaction between the producer and consumer is deterministic. What does deterministic mean? Assume for this question that if the OS_Fifo has 5 elements data is lost, but if it has 6 elements no data is lost. What does this tell you about the timing of the consumer plus display?

Deterministic means there is no uncertainty, and the order of execution of the Producer and Consumer is guaranteed. Producer puts data in a specified frequency no matter FIFO there is data lost or not. Consumer only passes through FIFO_Get when FIFO is not empty, which is controlled by producer. Display only passes through MailBox_Recv if Consumer has sent data.

Consumer runs at relatively the same frequency (when reaching the end of the loop, there are some extra data to process hence some extra execution time, also other threads' execution times may vary). If this frequency is sufficiently high, Producer will never fill up the FIFO. And for a given frequency, the size of the FIFO determines whether it's ever going to be filled up.

To answer this questions, the execution rate of consumer plus display is at a rate that is too slow compared to producer when there are 5 slots in FIFO, but not too slow when there are 6. I believe Display only adds another context switch delay and execution time to the period of Consumer’s execution. Display never blocks Consumer’s action in round-robin scheduling system.

7) Without going back and actually measuring it, do you think the Consumer ever waits when it calls OS_MailBox_Send? Explain.

No. The way this system is configured, for every time OS_MailBox_Send is called, OS_MailBox_Recv is called in the upcoming Display thread. Therefore, there will never be a situation where OS_MailBox_Send is called consecutively without Display signaling the DataFree semaphore in between.

This is mainly due to the low ADC input rate of Producer, causing Consumer to hold a lot.