Lab 3: RTOS with Blocking and Priority Scheduling

Xinyuan (Allen) Pan & Qinyu (Jack) Diao

Objective

The objective of this lab was to extend the RTOS kernel we developed in lab 2. We coordinated multiple foreground and background threads using a priority scheduler and blocking preemptive semaphores for thread synchronization. The threads communicated with each other through FIFOs and a mailbox. We also include the support for two real-time periodic tasks.

Software Design

```
unsigned long NumSamples;
unsigned long maxJitter1;   // in 0.1us units
unsigned long maxJitter2;
unsigned long jitter1Histogram[JITTERSIZE]={0,};
unsigned long jitter2Histogram[JITTERSIZE]={0,};

void print_jitter(void) {
//     ST7735_Message(1,0,"Jitter 1 = ", maxJitter1);
//     ST7735_Message(1,1,"Jitter 2 = ", maxJitter2);
       Serial_println("Periodic Task 1 jitter (0.1 us): %u", maxJitter1);
       Serial_println("Periodic Task 2 jitter (0.1 us): %u", maxJitter2);
}

void Timer1A_Handler(void){
       static unsigned long lastTime;
       unsigned long jitter;
       TIMER1_ICR_R = TIMER_ICR_TATOCINT;  // acknowledge
       unsigned long thisTime;

       if (NumSamples < RUNLENGTH) {
              thisTime= OS_Time();       // current time, 12.5 ns

              periodic_tasks[0]();                // execute user task
              periodic_counters[0]++;
              if(periodic_counters[0]>1){    // ignore timing of first interrupt
                     unsigned long diff = OS_TimeDifference(lastTime, thisTime);
                     if (diff > periodic_periods[0])
                            jitter = (diff-periodic_periods[0]+4)/8;  // in 0.1 usec
                     else
                            jitter = (periodic_periods[0]-diff+4)/8;  // in 0.1 usec
                     if(jitter > maxJitter1)
                            maxJitter1 = jitter; // in usec
                     // jitter should be 0
```

```
                if(jitter >= JITTERSIZE)
                        jitter = JITTERSIZE-1;
                jitter1Histogram[jitter]++;
            }
            lastTime = thisTime;
        }
}

void Timer0A_Handler(void){
        static unsigned long lastTime;
        unsigned long jitter;
        TIMER0_ICR_R = TIMER_ICR_TATOCINT;  // acknowledge
        unsigned long thisTime;
        if (NumSamples < RUNLENGTH) {
                thisTime = OS_Time();       // current time, 12.5 ns
                periodic_tasks[1]();                // execute user task
                periodic_counters[1]++;
                if(periodic_counters[1]>1){    // ignore timing of first interrupt
                        unsigned long diff = OS_TimeDifference(lastTime, thisTime);
                        if (diff > periodic_periods[1])
                                jitter = (diff-periodic_periods[1]+4)/8;  // in 0.1 usec
                        else
                                jitter = (periodic_periods[1]-diff+4)/8;  // in 0.1 usec
                        if(jitter > maxJitter2)
                                maxJitter2 = jitter; // in usec
                        // jitter should be 0
                        if(jitter >= JITTERSIZE)
                                jitter = JITTERSIZE-1;
                        jitter2Histogram[jitter]++;
                }
                lastTime = thisTime;
        }
}

// ******** OS_InitSemaphore ************
// initialize semaphore
// input:  pointer to a semaphore
// output: none
void OS_InitSemaphore(Sema4Type *semaPt, long value) {
        semaPt->value = value;
        semaPt->start = 0;
        semaPt->end = 0;
}

/* Wait can only be called by main thread, because suspend (thread switch) only
applies to main threads */
// ******** OS_Wait ************
// decrement semaphore
// Lab2 spinlock
// Lab3 block if less than zero
```

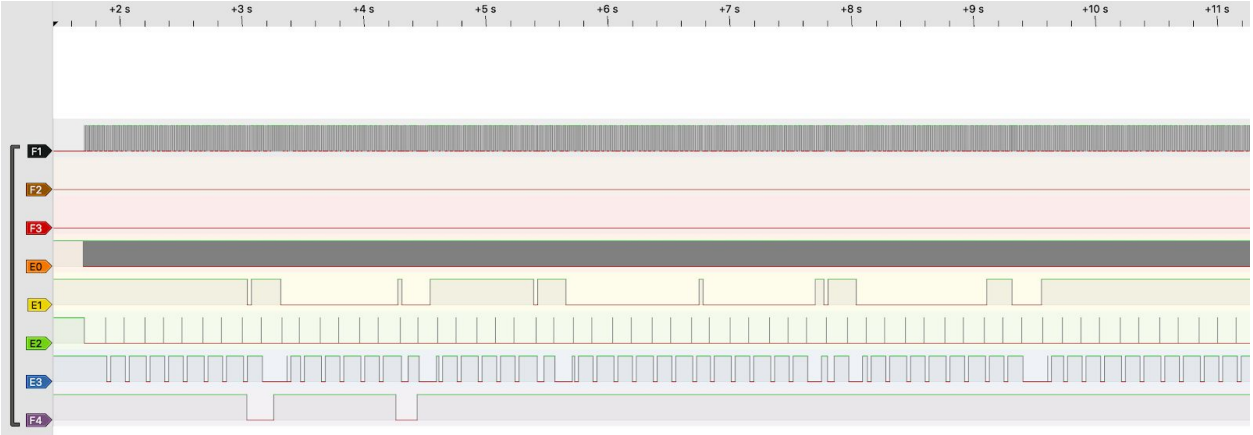
```
// input:  pointer to a counting semaphore
// output: none
void OS_Wait(Sema4Type *semaPt) {
	OS_DisableInterrupts();
	semaPt->value = semaPt->value - 1;
	if (semaPt->value < 0) {
		RunPt->state = BLOCKED;
		RunPt->blocked = semaPt;
		semaPt->waiters[semaPt->end] = RunPt;  // add to waiters list
		semaPt->end = (semaPt->end + 1) % NUMTHREADS;
		OS_EnableInterrupts();
		OS_Suspend();
	}
	OS_EnableInterrupts();
}

// ******** 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) {
	unsigned long sr = StartCritical();
	tcbType *pt = RunPt;
	semaPt->value = semaPt->value + 1;
	if (semaPt->value <= 0) {
		semaPt->waiters[semaPt->start]->state = ACTIVE;		// release
the first blocked thread
		semaPt->start = (semaPt->start + 1) % NUMTHREADS;
	}
	EndCritical(sr);
}


// ******** 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_DisableInterrupts();
	while (semaPt->value == 0) {
		RunPt->state = BLOCKED;
		RunPt->blocked = semaPt;
		semaPt->waiters[semaPt->end] = RunPt;  // add to waiters list
		semaPt->end = (semaPt->end + 1) % NUMTHREADS;
		OS_EnableInterrupts();
		OS_Suspend();
	}
```

```
        semaPt->value = 0;    // write zero back to it, prepared for usage next time
        OS_EnableInterrupts();
}

// ******** 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) {
    unsigned long sr = StartCritical();  // why save I bit here?
    if (semaPt->value == 0) {
       semaPt->waiters[semaPt->start]->state = ACTIVE;             // release the
first blocked thread
       semaPt->start = (semaPt->start + 1) % NUMTHREADS;
    }
       semaPt->value = 1;
    EndCritical(sr);
}

// schedules the next thread to run
void threadScheduler(void) {
       tcbType * pt = RunPt;
       tcbType * endPt;  // endPt is the last thread to check in the Linked List
       // whether this thread is killed
       if (RunPt->state == FREE) {
              endPt = RunPt->prev;
       }
       else {
              endPt = RunPt;
       }
       tcbType * bestPt;
       int maxPri = 255;

       do {
              pt = pt->next;
              if (pt->state == ACTIVE && pt->priority < maxPri) {
                     maxPri = pt->priority;
                     bestPt = pt;
              }
       } while (pt != endPt);
       RunPt = bestPt;

#ifdef DEBUG
//     LED_BLUE_TOGGLE();
#endif

enum State {
              FREE,
              ACTIVE,
```

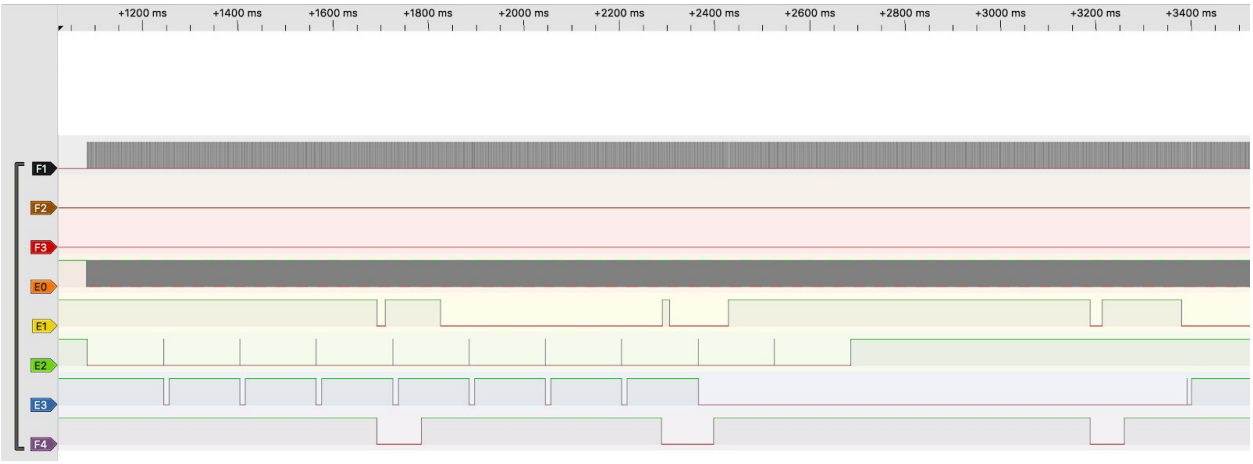
```
            SLEEP,
            BLOCKED
};

typedef struct tcb tcbType;

typedef struct Sema4{
  long value;   // > 0 means free, otherwise means busy
  tcbType *waiters[NUMTHREADS];  // the threads waiting on this sema
  int start;                     // the start index of waiters
  int end;                                    // the end index of waiters
} 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;         // the semaphore it is blocked on
       int32_t priority;
} tcbType;
```

Measurement Data

1) Plot of the logic analyzer running the blocking/sleeping/killing/round-robin system



2) Plot of the scope window running the blocking/sleeping/killing/priority system



3) Table like Table 3.1 each showing performance measurements versus sizes of the Fifo and timeslices

		Spinlock/Round Robin			Blocking/Round Robin			Blocking/Priority		
FIFO Size	TSlice (ms)	Data Lost	Jitter (µs)	PID Work	Data Lost	Jitter (µs)	PID Work	Data Lost	Jitter (µs)	PID Work
2	2	3182	1.0	4964	3182	1.0	5238	361	3.1	8173
16	2	0	0.9	4842	0	1.1	5110	0	3.0	8151
16	1	0	1.0	4834	0	1.4	5104	0	3.0	6046
16	10	0	0.9	4837	0	1.1	5107	0	2.9	10160

Analysis and Discussion

1) How would your implementation of OS_AddPeriodicThread be different if there were 10 background threads? (Preparation 1)

Because the number of the background threads is more than the available timers, the background threads have to be scheduled by OS and triggered by scheduler. To implement priority for background threads, we can use negative priorities so that background threads are always prioritized over foreground threads.

2) How would your implementation of blocking semaphores be different if there were 100 foreground threads? (Preparation 4)

Our current implementation of blocking semaphores is actually suitable for any number of threads, because each semaphore has its own queue of blocked threads. There is no need to search for which thread to unblock in signal.

3) How would your implementation of the priority scheduler be different if there were 100 foreground threads? (Preparation 5)

There are a lot of implementations that can be used in place of our solution. The main idea is (1) not to loop through every thread in the scheduler (which is what is done currently), and (2) to avoid starvation of the threads with low priority. Multi-level Feedback Queue will be one of the solutions that accomplishes this.

4) What happens to your OS if all the threads are blocked? If your OS would crash, describe exactly what the OS does? What happens to your OS if all the threads are sleeping? If your OS would crash, describe exactly what the OS does? If you answered crash to either or both, explain how you might change your OS to prevent the crash.

If all the threads are blocked or sleeping, the OS will crash because nothing is active to run. In our implementation, RunPt will be assigned to 0, and the system will likely run into exceptions by trying to read from there as if it's a tcb. The behaviors are the same for sleep and blocked in our implementation.

We can create an idle task and force the OS to run the idle task (only runs if nothing else is active) if the OS cannot find anything else to execute.

5) What happens to your OS if one of the foreground threads returns? E.g., what if you added this foreground

```
void BadThread(void)
{ int i; for(i=0; i<100; i++){}; return; }
```

What should your OS have done in this case? Do not implement it, rather, with one sentence, say what the OS should have done?

Once a foreground thread returns, it will execute BX LR. If in the code LR is always correctly saved and restored when making function calls, it should contain the initialized value, which we had set to 0x14141414, upon return. The OS should have initialized LR to the address of a special function that handles this situation, e.g. simply calling OS_KILL.

6) What are the advantages of spinlock semaphores over blocking semaphores? What are the advantages of blocking semaphores over spinlock?

Advantages of spinlock semaphores over blocking:

a) Fewer data structures, therefore require less memory.
b) Lower thread switching overhead (although largely offset by wasted cycles)

Advantages of blocking semaphores over spinlock:

a) To recapture the lost processing time wasted by scheduling a thread that can't do useful work.
b) To implement bounded waiting to set a limit of how many threads can run before a certain thread, hence solving the fairness issue.
c) Can be implemented to choose the thread to be released based on priorities.

7) Consider the case where thread T1 interrupts thread T2, and we are experimentally verifying the system operates without critical sections. Let *n* be the number of times T1 interrupts T2. Let *m* be the total number of interruptible locations within T2. Assume the time during which T1 triggers is random with respect to the place (between which two instructions of T2) it gets interrupted. In other words, there are *m* equally-likely places within T2 for the T1 interrupt to occur.

What is the probability after n interrupts that a particular place in T2 was never selected?

$$\left(\frac{m-1}{m}\right)^n$$

Furthermore, what is the probability that all locations were interrupted at least once?

$$\binom{n-1}{m-1} = \frac{(n-1)!}{(m-1)!\,(n-m+1)!}$$