EE 445M: Lab 03_Priority Scheduler and Blocking Semaphore

Tejasree Ramanuja (td23239) and Chioma Okorie (coo279)

03/08/18

Objectives

The goal of this lab is to extend Lab2 to use a priority scheduler and to implement blocking semaphores. The blocking semaphore allows us to implement bounded waiting and the priority scheduler allows us to service threads with higher priority first. These two improvements to the previous lab allows the scheduler to only run important tasks. We also extended the RTOS to include two periodic and two aperiodic tasks. In addition, we took some performance measurements of our system.

Hardware Design

No hardware design required. We used the ST7735 display, TM4C123 microcontroller, and the robot sensor board.

Software Design

1. **Documentation and code of main program used to measure time jitter in Procedure 2**

   Main program used for Jitter Calculations are the same at the Lab2.c file provided to us. We did not modify it. Below is the function that we wrote to calculate the jitter. We have also included our jitter measurements for procedure 2.

   *****************Jitter Calculation******************

```
void Timer1A_Handler(void){
        unsigned long thisTime = OS_Time();
        unsigned static long LastTime;
        long jitter;

  TIMER1_ICR_R = TIMER_ICR_TATOCINT;// acknowledge TIMER1A
timeout
if (DumpIndex < DumpSize){
        Dump[DumpIndex].event_periodic = 2; // periodic 1
        Dump[DumpIndex].timestamp = OS_Time();
        DumpIndex++;
}
  (*PeriodicThread2)();              // execute user task
if (DumpIndex < DumpSize){
```

```
        Dump[DumpIndex].event_periodic = 2; // periodic 1
        Dump[DumpIndex].timestamp = OS_Time();
        DumpIndex++;
}

#if Lab3
    if (PerTask2Counter){
        unsigned long diff = OS_TimeDifference(LastTime,thisTime);
     if(diff>PerTask2Period){
      jitter = (diff-PerTask2Period+4)/8;  // in 0.1 usec
     }else{
      jitter = (PerTask2Period-diff+4)/8;  // in 0.1 usec
     }
     if(jitter > MaxJitter2){
      MaxJitter2 = jitter; // in usec
     }       // jitter should be 0
     if(jitter >= Jitter_Size){
      jitter = JITTER_SIZE-1;
     }
     JitterHistogram2[jitter]++;
    }
         LastTime = thisTime;
         PerTask2Counter++;
#endif
}
void Timer4A_Handler(void){
        unsigned long thisTime = OS_Time();
        unsigned static long LastTime;
        long jitter;

  TIMER4_ICR_R = TIMER_ICR_TATOCINT;       // acknowledge TIMER4A
timeout
        if (DumpIndex < DumpSize){
                Dump[DumpIndex].event_periodic = 1; // periodic 1
                Dump[DumpIndex].timestamp = OS_Time();
                DumpIndex++;
        }
  (*PeriodicThread1)();              // execute user task
```

```
if (DumpIndex < DumpSize){
        Dump[DumpIndex].event_periodic = 1; // periodic 1
        Dump[DumpIndex].timestamp = OS_Time();
        DumpIndex++;
}

#if Lab3
        if (PerTask1Counter){
                unsigned long diff = OS_TimeDifference(LastTime,thisTime);
    if(diff>PerTask1Period){
     jitter = (diff-PerTask1Period+4)/8;  // in 0.1 usec
    }else{
     jitter = (PerTask1Period-diff+4)/8;  // in 0.1 usec
    }
    if(jitter > MaxJitter1){
     MaxJitter1 = jitter; // in usec
    }       // jitter should be 0
    if(jitter >= Jitter_Size){
     jitter = JITTER_SIZE-1;
    }
    JitterHistogram1[jitter]++;
        }
         LastTime = thisTime;
         PerTask1Counter++;
#endif
}
```

************************Jitter Measurements**********************

Task A priority 0 & Task B priority 1

Task A (execution time 500, 1ms period)

- Periodic_Task1 Jitter 0.1 us = 1
- Periodic_Task2 Jitter 0.1 us = 1

Task A (execution time 500, 2.9999625 ms period)

- Periodic_Task1 Jitter 0.1 us = 0
- Periodic_Task2 Jitter 0.1 us = 6

Task A (execution time 50, 1ms period)

- Periodic_Task1 Jitter 0.1 us = 1
- Periodic_Task2 Jitter 0.1 us = 1

Task A (execution time 50, 2.9999625 ms period)

- Periodic_Task1 Jitter 0.1 us = 0
- Periodic_Task2 Jitter 0.1 us = 6

Task A (execution time 5, 1ms period)

- Periodic_Task1 Jitter 0.1 us = 0
- Periodic_Task2 Jitter 0.1 us = 1

Task A (execution time 5, 2.9999625 ms period)

- Periodic_Task1 Jitter 0.1 us = 0
- Periodic_Task2 Jitter 0.1 us = 7

2. **Documentation and code of main program used test the blocking semaphores Preparation 4**

Main program used to test blocking semaphore is the same as the one provided in Lab3.c (lab 3 preparation 4). Here are the functions that we wrote to implement the blocking semaphore.

************************Blocking Semaphore************************

```
void UnchainTCB(void){
        RunPt->prev->next = RunPt->next;
        RunPt->next->prev = RunPt->prev;
        NumThreads--;
}
void ChainTCB(tcbType *newActiveThread){
        tcbType *last = RunPt->prev;
        Pri_Available[newActiveThread->priority] =
        Pri_Available[newActiveThread->priority] + 1;
        newActiveThread->next = RunPt;
        RunPt->prev = newActiveThread;
        newActiveThread->prev = last;
        last->next = newActiveThread;
        NumThreads++;
}
void AddBlockedTCB_Sema4(Sema4Type *semaPt){
        //find the end of the blocked list
        if(semaPt->blockedThreads == 0){//add head
                semaPt->blockedThreads = RunPt;
                semaPt->blockedThreads->nextBlocked = 0; //current thread is the
                end of the blocked list
        }
        else{//find the end of the list for the tcbs blocked on the sema4
```

```
                tcbType *tail = semaPt->blockedThreads;
                while(tail->nextBlocked != 0){
                        tail = tail->nextBlocked;
                }
                tail->nextBlocked = RunPt;
                tail->nextBlocked->nextBlocked = 0;
        }
}

tcbType* RemoveBlockedTCB_sema4(Sema4Type *semaPt){// assumes that
at least one thread is blocked on this sema4
        tcbType *head = semaPt->blockedThreads;
        semaPt->blockedThreads = head->nextBlocked;
        return head;
}

void Block(Sema4Type *semaPt){
        RunPt->blocked = semaPt;
        Pri_Available[RunPt->priority] = Pri_Available[RunPt->priority] - 1;
        UnchainTCB(); // remeove current tcb from active list (does not free tcb)
        AddBlockedTCB_Sema4(semaPt);
        OS_Suspend();
}

void UnBlock(Sema4Type *semaPt){
        //remove the first tcb blocked on this sema4 from the blocked list
        tcbType *blockedTCB = RemoveBlockedTCB_sema4(semaPt);
        //add tcb to the active list
        ChainTCB(blockedTCB);
        blockedTCB->blocked = 0; //wake up threads
}

void OS_Wait(Sema4Type *semaPt){
        long status = StartCritical();
        semaPt->Value = semaPt->Value - 1;
        if(semaPt->Value < 0){
                Block(semaPt);
        }
        EndCritical(status);
```

```
}

void OS_Signal(Sema4Type *semaPt){
        long status = StartCritical();
        semaPt->Value = semaPt->Value + 1;
        if(semaPt->Value <= 0){
                UnBlock(semaPt);
        }
        EndCritical(status);
}

void OS_bWait(Sema4Type *semaPt){
        OS_DisableInterrupts();
        (*semaPt).Value -=1;
        if(semaPt->Value < 0){
                Block(semaPt);
        }
        OS_EnableInterrupts();
}

void OS_bSignal(Sema4Type *semaPt){
        int32_t status = StartCritical();
        semaPt->Value += 1;
        if(semaPt->Value <= 0){
                UnBlock(semaPt);
        }
        EndCritical(status);
}
```

3. **Documentation and code of your blocking/priority RTOS, OS.c and any associated assembly files**

```
**********Add Priority Thread**********
void AddPriThread (tcbType *thread){
        uint8_t priority = thread->priority;

        if(Pri_Total[priority] == 0){
                PriPt[priority] = thread;
                PriLastPt[priority] = thread;
                thread->Pri_Next = thread;
        } else {        // Add it to the beginning of the priority "linked list"
                thread->Pri_Next = PriLastPt[priority]->Pri_Next;
                PriLastPt[priority]->Pri_Next = thread;
                PriPt[priority] = thread;
        }

        Pri_Total[priority]++; Pri_Available[priority]++;
}

**********OS_Kill**********
        void OS_Kill(void){
        #if !PriScheduler
                OS_DisableInterrupts();
                UnchainTCB();  //remove current running thread from the circular linked list
                tcbs[RunPt->Id].status = -1; // remember which tcb is free
                OS_Suspend(); //switch to the next task to run
                OS_EnableInterrupts();
                for(;;){}
        #else
                OS_DisableInterrupts();
                tcbType *temp, *current;
                if ((RunPt == PriPt[RunPt->priority]) && (RunPt ==
                PriLastPt[RunPt->priority])){ // only one node in the list
                        PriPt[RunPt->priority] = 0;
                        PriLastPt[RunPt->priority] = 0;
                } else if (RunPt == PriPt[RunPt->priority]){ // Delete the first node in the list
```

```
                PriLastPt[RunPt->priority]->Pri_Next = RunPt->Pri_Next;
                PriPt[RunPt->priority] = PriLastPt[RunPt->priority]->Pri_Next;
        } else {
                        temp = current = PriPt[RunPt->priority];
                        while (current != RunPt){
                                temp = current; current = current->Pri_Next;
                }
                if (RunPt == PriLastPt[RunPt->priority]){
                temp->Pri_Next = PriPt[RunPt->priority];
                        PriLastPt[RunPt->priority] = temp;
                } else {        // Delete a specific node in the list
                        temp->Pri_Next = current->Pri_Next;
                }
        }
        //RunPt->Pri_Next = 0;
        tcbs[RunPt->Id].status = -1;
        NumThreads--;
        Pri_Total[RunPt->priority] = Pri_Total[RunPt->priority] - 1;
        Pri_Available[RunPt->priority] = Pri_Available[RunPt->priority] - 1;
        OS_Suspend(); //switch to the next task to run
        OS_EnableInterrupts();
        for(;;){}
#endif
}
```

******************************OS_Sleep*************************

```
void OS_Sleep(unsigned long sleepTime){
        OS_DisableInterrupts();
        RunPt ->sleep = sleepTime;
        #if PriScheduler
        Pri_Available[RunPt->priority] = Pri_Available[RunPt->priority] - 1;
        #endif
        OS_Suspend();
        OS_EnableInterrupts();
}
```

Measurement Data

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

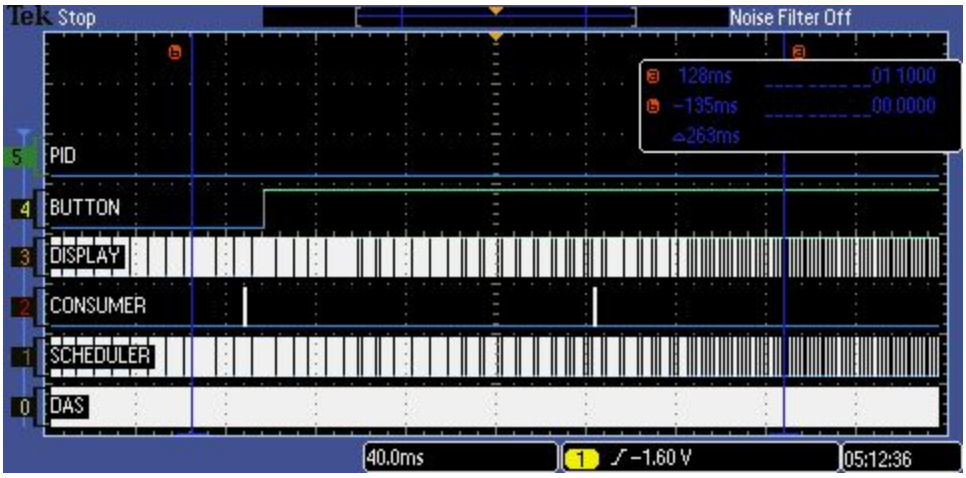

Figure 1: Scope picture of the the blocking/sleeping/killing/round-robin system

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

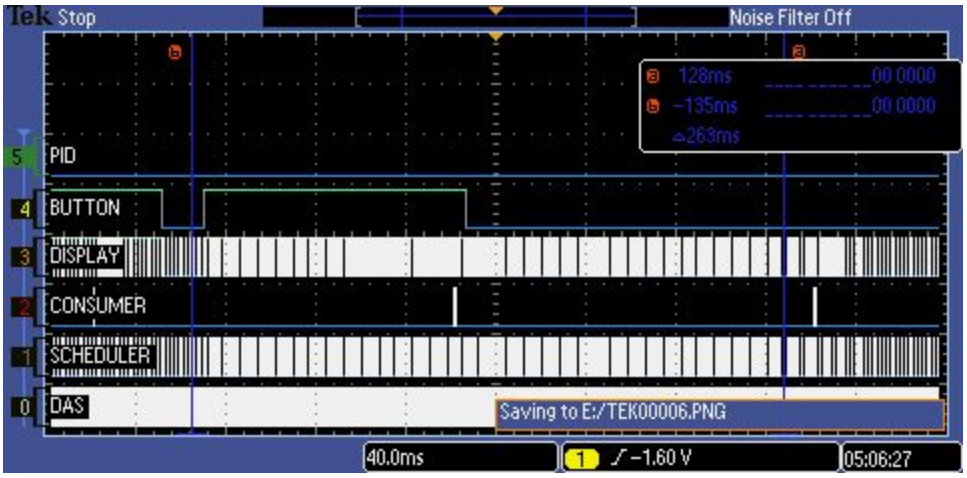

Figure 2: Scope window with PID running and one of the buttons pressed

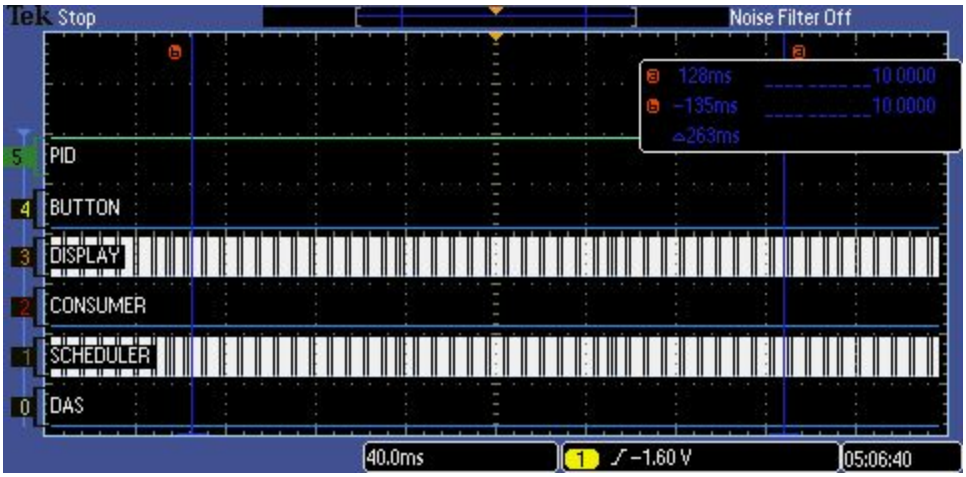

Figure 3: Scope picture with PID work done and neither button pressed

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

		Spinlock/Round Robin			**Spinlock/Round Robin/Cooperative**			**Blocking/Priority**		
FIFO size	**Tslice (ms)**	**Data lost**	**Jitter (μs)**	**PID work**	**Data lost**	**Jitter (μs)**	**PID work**	**Data lost**	**Jitter (μs)**	**PID work**
4	2	0	0.3	3335	0	0.6	6152	1	2.1	9818
32	2	0	0.3	3335	0	0.6	6151	0	1.9	9817
32	1	0	0.3	3333	0	0.5	6142	0	2.2	7333
32	10	0	0.3	3337	0	0.4	6159	0	1.9	12272

Analysis and Discussion

1. **How would your implementation of OS_AddPeriodicThread be different if there were 10 background threads?**

   First, we find the LCM of the period of the periodic tasks that we need to run.

We set a count to count up to the LCM and reset. For each periodic task, we schedule it at the desired time but non-overlapping with the other periodic tasks that are scheduled. This periodic tasks can be run in the scheduler or using a separate timer. To ensure that the system operates at desired, we must schedule at most one periodic task every SysTick ISR execution. Also the time it takes to execute these tasks should be much less than the time required for a thread switch to occur.

2. **How would your implementation of blocking semaphores be different if there were 100 foreground threads?**
   In this lab, we implemented blocking semaphore by removing a thread that is blocked from the active list. This will ensure that the scheduler only deals with tasks that are active instead to traversing tcbs with threads that are not ready to run. The only thing that we might do differently is to remove sleeping threads from the active list as well.

3. **How would your implementation of the priority scheduler be different if there were 100 foreground threads?**
   In this lab, we also implemented priority scheduler in levels. Threads at the highest priority level gets run all the time as in a round robin scheduler. If all the threads in the higher priority level is either blocked or sleeping, the priority scheduler steps down one level and so on. If a thread at a lower priority gets scheduled because the threads in the higher priority were not ready to run, on the next context switch, the scheduler starts that the higher priority to look for thread to run. If we had more than 10 threads, it would be beneficial to add fields to the tcbs such as aging, working, and fixed priority to prevent tasks with lower priority from being starved of CPU time.

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.**

   Our implementation of blocking semaphore depends on the fact that at least one thread must be running at all times. If all the threads where blocked, the scheduler will not have any threads to schedule. The RunPt will be invalid and therefore the system would crash. More specifically, in Keil, the system would throw a Hard Fault Exception.

   Also, Since we did not unchain the tcb when a thread goes to sleep, the OS will keeping traversing the tcb until at least one of the threads wakes up.

5. **What happens to your OS if one of the foreground threads returns? E.g., what if you added this foreground. What should your OS have done in this case?**

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

All the threads that we have scheduled so far runs indefinitely until the thread is killed. In this case, the stack and tcb is freed. In this function, the system would throw a hard fault exception since a thread cannot return. To fix this, we can insert an OS_Kill function instead of the return.

6. **What are the advantages of spinlock semaphores over blocking semaphores? What are the advantages of blocking semaphores over spinlock?**

Advantages of Spinlock: Spinlock semaphore is easier to implement then the blocking semaphore.
Advantage of Blocking: the spinlock semaphore wastes a lot of cpu time. This is because the scheduler schedules a task that will not run which is inefficient. Implementing blocking allows us to remove threads that are not ready to run from the active lists until the resources are available. Blocking semaphores also allow us to implement bounded waiting so that thread that are waiting the longest get the resource first. Also, the priority scheduler requires the use of blocking semaphore since we do not want high priority tasks to spin preventing lower priority task from doing useful work.

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.
    a. **What is the probability after n interrupts that a particular place in T2 was never selected?**
    $m(\frac{m-1}{m})^n$
    b. **Furthermore, what is the probability that all locations were interrupted at least once?**

$$n < m \qquad P = 0$$
$$n \geq m \Rightarrow P = 1 - \sum_{i=1}^{m-1} P_i$$
$$\text{where} \quad P_i = [C_i^m]\left[\left(1-\frac{1}{m}\right)^i\left(\frac{1}{m}\right)^{n-i}\right] = [C_i^m]\left[\frac{(m-i)^i}{m^n}\right]$$