Sam Caldwell
Rohith Prakash

Lab 3 Report

Objectives:

In this lab, we modified our operating system (OS) to use a priority scheduler and blocking semaphores. Having a priority scheduler instead of round-robin allows us to give precedence to I/O bound tasks. Blocking semaphores reduce time wasted polling a semaphore. This lab also asked us to introduce a second periodic task with configurable priority. However, our OS already supported this. We also added more powerful logging and profiling measures and corresponding shell commands to help us debug and measure the OS. Additionally, we added an interrupt for the Down button which has the same functionality as the previously implemented Select button to run a user-specified thread.

Software Design:

Jitter measurement program:

```
int main(void)
{
  Init_Everything();

  //********initialize communication channels
  OS_MailBox_Init();
  OS_Fifo_Init(32);

  NumCreated = 0;
  NumSamples = 0;
//   MaxJitter = 0;      // OS_Time in 20ns units
//   MinJitter = 10000000;

  #if PROFILING
    // intialize port b pins as specified by PINS mask
    // for digital output for use in profiling threads
    SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOB;
    delay = SYSCTL_RCGC2_R;
    GPIO_PORTB_DIR_R |= PINS;
    GPIO_PORTB_DEN_R |= PINS;
    GPIO_PORTB_DATA_R &= ~PINS;
  #endif

  // testing/debugging stuff
  DASPeriodicID = OS_Add_Periodic_Thread(&DAS,2,1); // .5 kHz real time sampling (breaks project
currently)
//  OS_AddButtonTask(&dummyButtonTask, 1);
  OS_AddButtonTask(&ButtonPush, 1);
  OS_AddDownTask(&dummyDownTask, 1);
```

```
  NumCreated += OS_AddThread(&jerkTask, 0, 6);
  NumCreated += OS_AddThread(&dummyTask3, 0, 7);
  NumCreated += OS_AddThread(&dummyTask1, 0, 2);
  NumCreated += OS_AddThread(&PID, 0, 3);
  NumCreated += OS_AddThread(&dummyTask2, 0, 2);
  NumCreated += OS_AddThread(&Consumer, 128, 3);
  NumCreated += OS_AddThread(&SH_Shell, 0, 3);
  OS_Launch(TIMESLICE);

  /* Loop indefinitely */
  while(1);
}
```

Blocking semaphore test program:

```
int main(void)
{
  Init_Everything();

  //********initialize communication channels
  OS_MailBox_Init();
  OS_Fifo_Init(32);

  NumCreated = 0;
  NumSamples = 0;
  MaxJitter = 0;      // OS_Time in 20ns units
  MinJitter = 10000000;

  #if PROFILING
    // intialize port b pins as specified by PINS mask
    // for digital output for use in profiling threads
    SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOB;
    delay = SYSCTL_RCGC2_R;
    GPIO_PORTB_DIR_R |= PINS;
    GPIO_PORTB_DEN_R |= PINS;
    GPIO_PORTB_DATA_R &= ~PINS;
  #endif

  // testing/debugging stuff
  OS_Add_Periodic_Thread(&DAS,2,1); // .5 kHz real time sampling (breaks project currently)
//  OS_AddButtonTask(&dummyButtonTask, 1);
  OS_AddButtonTask(&ButtonPush, 1);
  OS_AddDownTask(&dummyDownTask, 1);

  NumCreated += OS_AddThread(&jerkTask, 0, 6);
  NumCreated += OS_AddThread(&dummyTask3, 0, 7);
  NumCreated += OS_AddThread(&dummyTask1, 0, 2);
```

```
  NumCreated += OS_AddThread(&PID, 0, 3);
  NumCreated += OS_AddThread(&dummyTask2, 0, 2);
  NumCreated += OS_AddThread(&Consumer, 128, 3);
  NumCreated += OS_AddThread(&SH_Shell, 0, 3);
  OS_Launch(TIMESLICE);

  /* Loop indefinitely */
  while(1);
}
```

Blocking/priority functions:

```
// block the currently running thread on a semaphore by setting its blocked flag to true
// and adding it to that semaphore's fifo
void OS_BlockThread(OS_SemaphoreType *s) {
  _TCB *thread = _RunPt;
  thread->block = 1;
  // semaphore fifo guaranteed to have room
  s->blockedThreads[s->PutIndex] = thread;
  s->PutIndex = (s->PutIndex + 1) & (_OS_MAX_THREADS - 1);
  OS_Suspend();
  OS_Delay(OS_ARBITRARY_DELAY);
}

// wake up 1 thread from the semaphore's fifo if there are any waiting
void OS_WakeThread(OS_SemaphoreType *s) {
  if(s->PutIndex != s->GetIndex) {
    s->blockedThreads[s->GetIndex]->block = 0; // wake up thread
    s->GetIndex = (s->GetIndex + 1) & (_OS_MAX_THREADS - 1);
  }
}

// wake up all threads blocked on a semaphore
void OS_WakeAllThreads(OS_SemaphoreType *s) {
  while(s->GetIndex != s->PutIndex) {
    OS_WakeThread(s);
  }
}

// during a context switch, index _RunPt to the first non-sleeping/blocked thread
void OS_FindNextThread(void)
{
  _TCB *temp, *head;
  // if last thread did not kill itself, remove it and re-insert it into proper position in LL
  if(_RunPt != NULL) {
    _RunPt->priority = _RunPt->base_priority; // reset priority
    _RunPt->run = 1; // mark as having executed
    _OS_RemoveFromLL(_RunPt);
```

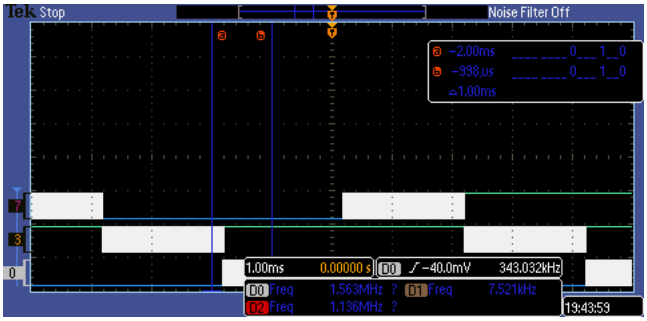
```
  _OS_InsertThread(_RunPt);
 }
 // find the next thread to execute
        temp = _TCBHead;
 // starting with highest priority, index forward to the first non-blocked,
 // non-sleeping thread
        while((temp->next != NULL) && (temp->block || temp->sleep))
                temp = temp->next;
 if(temp != NULL) {
  _RunPt = temp;
 }
 else {
  // TODO - add an OS_IllegalState handler?
 }
}

// increment priority of threads that haven't run after NUM_THREADS context switches
void OS_IncPriority(void)
{
        _TCB *temp = _TCBHead;
        static int num = 0;
        if(++num % _OS_MAX_THREADS)
                return;
        while(temp != NULL)
        {
                if(!temp->run && temp->priority > 0) {
                        temp->priority--;
    // remove and re-insert incase chaning priority changed its position in LL
    _OS_RemoveFromLL(temp);
    _OS_InsertThread(temp);
   }
   else {
    // reset run flag for next round
    temp->run = 0;
   }
                temp = temp->next;
        }
}

// Insert a thread into it's proper position in the LL
// either because it's priority has changed or it has run
// assume that thread is not currently in the LL
// much easier if LL is doubly linked
void _OS_InsertThread(_TCB* thread) {
 _TCB *temp = _TCBHead;
 // find correct position in LL
 if(temp == NULL) {
  // LL is currently empty
```

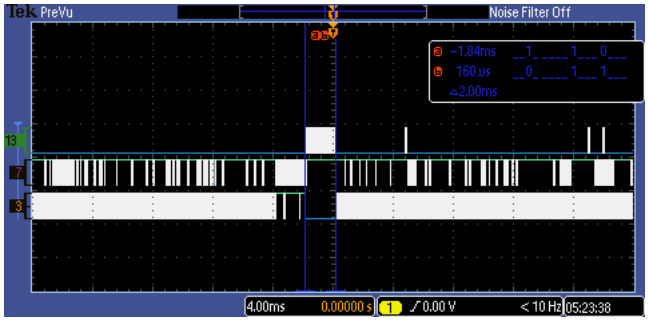
```
   _TCBHead = thread;
   thread->next = thread->prev = NULL;
  }
  else if(thread->priority < temp->priority) {
   // higher priority than the head so put at beginning of LL
   thread->next = temp;
   temp->prev = thread;
   thread->prev = NULL;
   _TCBHead = thread;
  }
  else {
   // this thread should become the last of its priority
   while((temp->next != NULL) && (temp->next->priority <= thread->priority)) {
    temp = temp->next;
   }
   // place between temp and temp->next
   thread->next = temp->next;
   thread->prev = temp;
   if(temp->next != NULL) {
    // test in case adding at end of LL
    temp->next->prev = thread;
   }
   temp->next = thread;
  }
}

// remove a thread from the LL
void _OS_RemoveFromLL(_TCB *thread) {
  if(_TCBHead == thread) {
   // make thread->next the new head
   _TCBHead = thread->next;
   _TCBHead->prev = NULL;
  }
  else {
   thread->prev->next = thread->next;
   thread->next->prev = thread->prev;
  }
}

; void OS_bWait(OS_SemaphoreType* s)
; acquire binary semaphore s
OS_bWait
  ; decrement the number of permits
  LDREX R1, [R0]     ; R1 = value of *s (num permits)
  SUBS R1, #1        ; R1--, set condition codes
  STREX R2, R1, [R0]  ; loaded a non-zero value, so attempt to decrement and store
  CMP R2, #0
  BNE OS_bWait       ; repeat until successful decrement
```

```
  CMP R1, #0        ; if R1 >= 0, then have successfully acquired semaphore so can return
  IT PL
  BXPL LR
  ; block thread
  PUSH {LR}
  BL.W OS_BlockThread
  POP {LR}
  BX LR

; void OS_bSignal(OS_SemaphoreType* s)
; release binary semaphore s
OS_bSignal
  ; increment number of permits
  LDREX R1, [R0]
  ADD R1, R1, #1
  STREX R2, R1, [R0]
  CMP R2, #0
  BNE OS_bSignal    ; repeat until successful increment
  PUSH {LR}
  CMP R1, #0
  IT LE          ; if R1 <= 0, need to wake up a blocked thread
  BLLE OS_WakeThread
  POP {LR}
  BX LR

; void OS_Signal(OS_SemaphoreType* s)
; release a permit from semaphore s
; by incrementing value
OS_Signal
  ; increment number of permits
  LDREX R1, [R0]
  ADD R1, R1, #1
  STREX R2, R1, [R0]
  CMP R2, #0
  BNE OS_Signal    ; repeat until successful increment
  PUSH {LR}
  CMP R1, #0
  IT LE          ; if R1 <= 0, need to wake up a blocked thread
  BLLE OS_WakeThread
  POP {LR}
  BX LR

; void OS_Wait(OS_SemaphoreType* s)
; acquire a permit from semaphore s
; by decrementing value if greater than zero
OS_Wait
  ; decrement the number of permits
  LDREX R1, [R0]      ; R1 = value of *s (num permits)
```

```
SUBS R1, #1        ; R1--, set condition codes
STREX R2, R1, [R0]  ; loaded a non-zero value, so attempt to decrement and store
CMP R2, #0
BNE OS_Wait        ; repeat until successful decrement
CMP R1, #0         ; if R1 >= 0, then have successfully acquired semaphore so can return
IT PL
BXPL LR
; block thread
PUSH {LR}
BL.W OS_BlockThread
POP {LR}
BX LR
```

Measurement Data:





Task A period (ms):	1 2 4
Task A execution Time(us):	5 50 500
Task B period (ms):	2 2 2
Task B execution Time(us):	250 250 250
MinJitter (A):	-99 -187 -18
MaxJitter (A):	119 328 199
MinJitter (B):	-130 -82 -132
MaxJitter (B):	271 170 273

Spinlock/Round Robin					Block/Round Robin			Block/Priority		
Fifo Size	Tslice (ms)	Data Lost	Jitter (us)	PID Work	Data Lost	Jitter (us)	PID Work	Data Lost	Jitter (us)	PID Work
4	2	2993	8	2226	2893	6	2257	11	8	4539
8	2	0	8	2221	0	6	2252	0	14	4538
32	2	0	8	2221	0	6	2252	1	8	4538
32	1	0	8	2208	0	8	2238	17	14	4480
32	10	0	8	2228	0	6	2260	9	8	4580

Analysis and Discussion:

1) Our implementation would not change for 10 background threads, as our *OS_AddPeriodicThread* function has supported 10 background threads since Lab 1. To prevent jitter, we interrupt at an interval of *num_threads* * *min_period*. This way, we can stagger the scheduling of each task to avoid any task having to wait for another, as long as the periodic tasks do not take long to execute. Given that we set the minimum period at 1ms, this means that the hard deadline for a task to finish executing is 100us. Assuming no foreground tasks running, 10 tasks each taking 99us to execute would not affect the jitter of our system. However, most background tasks do not take this long to execute, so they do not adversely affect the foreground threads or other background tasks.
2) Currently in our OS, each semaphore object has data statically allocated for a queue of (pointers to) threads blocked on that semaphore. This queue needs to be large enough to support all foreground threads being blocked on any single semaphore. This would be much more expensive in terms of memory if there were 100 foreground threads. We could address this by changing there to be a single queue shared by all semaphores.
3) If there were 100 foreground threads, indexing through each thread in the linked list would take much longer, and we would not want to do so in the PendSV handler like we do now. Currently, we age threads which have not run once every *NUM_THREADS* iterations of the PendSV handler. If there are 100 threads, this might slow the handler more than desired, giving each thread less time to execute than is optimal. To avoid this, we would set Timer2B to do the task of aging every so often. We could have created a periodic task in Timer2A for this, but given that we have to worry about jitter there, it would be better to assign this to another timer altogether.
4) If all threads are blocked or sleeping (the scheduler doesn't make a distinction when looking for the next thread to run), the system will run a default thread added by the OS that that has the lowest priority and just calls *OS_Suspend* in a loop.
5) Our OS would crash if one of the foreground threads returned. There would be a stack underflow as there would be no address pushed onto the stack underneath for the PC to return to. There would be a value stored here, however, as the stack is statically located inside of the TCB structure. The PC would attempt to load the instruction at this location, but it would most likely not hold a valid instruction. This would then cause a hard fault. In this case, the OS should be able to detect a stack underflow and perform an *OS_Kill* on the thread in question. The system would then select the next thread to run.
6) An advantage of spin-lock semaphores is that they are simple to implement and require less overhead in terms of memory. Blocking semaphores reduce time spent polling a value that can't change until either an interrupt or another thread signals the semaphore.
7) Each event in this case is independent. As there are m equally likely places to interrupt, each interrupt has a $\frac{1}{m}$ chance of interrupting in a location *x*. After *n* trials, the probability that *x* has been interrupted at is $P(X \neq x)^c = P(X = x) = 1 - \left(\frac{m-1}{m}\right)^n$. Then, we see that the probability that *x* has never been selected is simply $P(X \neq x) = \left(\frac{m-1}{m}\right)^n$. Let *M* be the set of *m* locations.

To compute the probability that all locations in *M* have been interrupted at least once, we first examine the probability that any location in *M* has been interrupted. $\forall x \in M, P(X = x) = 1 - \left(\frac{m-1}{m}\right)^n$. The probability that any two locations are selected is the same, so for each location, the probability is the same. Therefore, the overall probability is $P(X = x; \forall x \in M) = \left(1 - \left(\frac{m-1}{m}\right)^n\right)^m$ ■