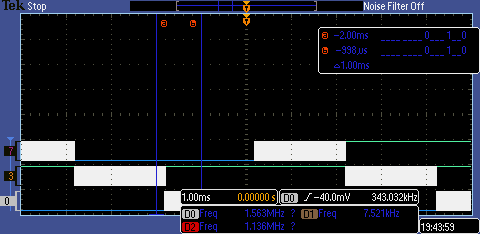
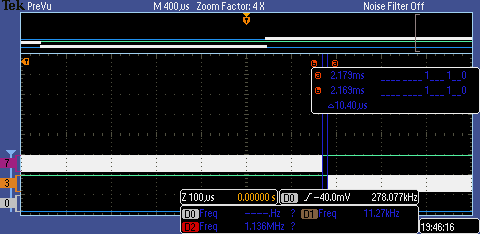
Sam Caldwell

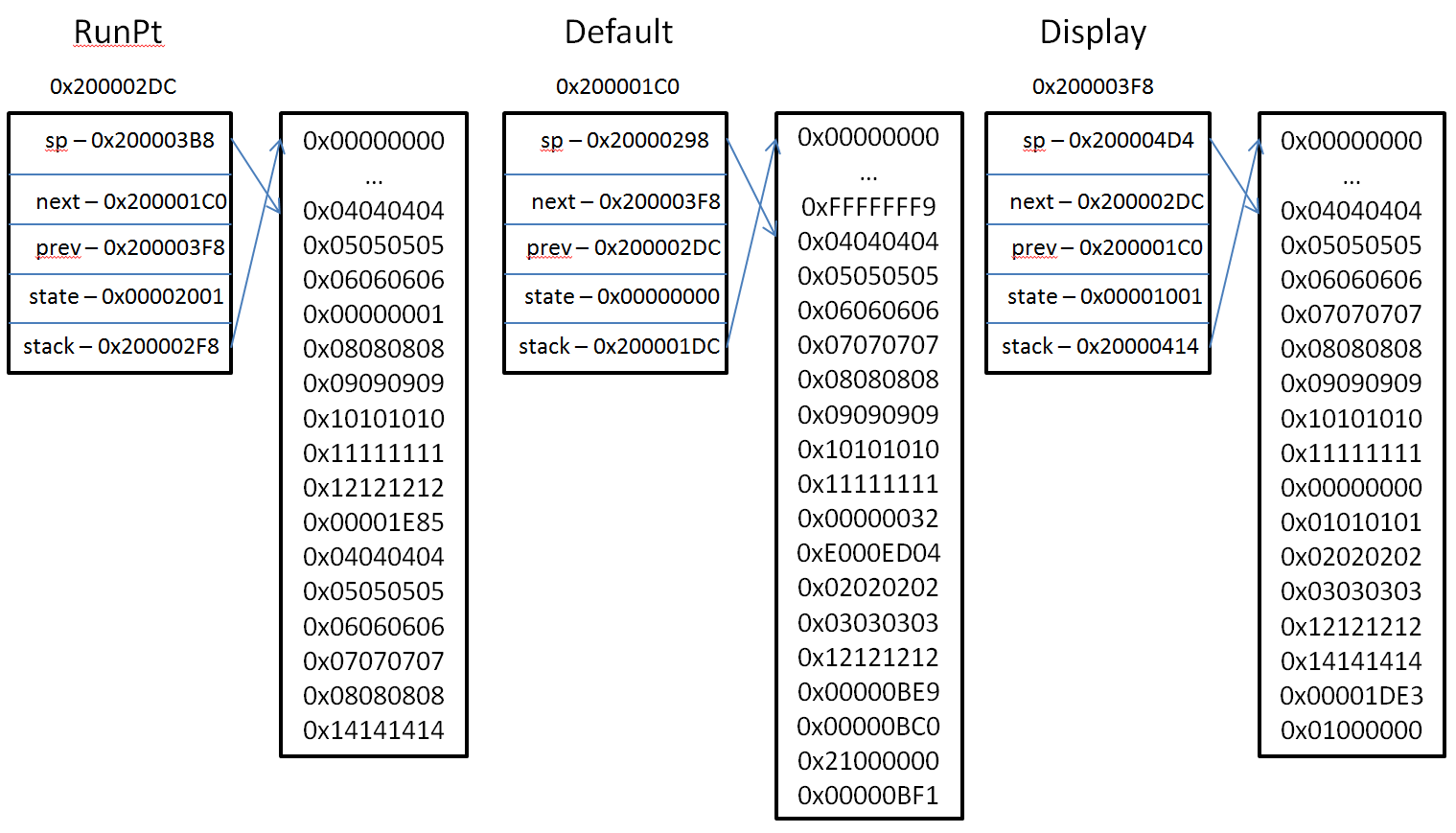
Rohith Prakash

Lab 2 Report

1. **Objectives**

The objective of this lab was to design a system which is capable of managing multiple foreground and background threads simultaneously, keeping each thread’s stack space and state separate. Additionally, this system must implement binary and counting semaphores as well as provide a method for foreground threads to sleep, kill, and suspend themselves. Lastly, this system must be able to provide a way for threads to communicate with each other. A fifo queue which supports a fixed number of elements and a mailbox which supports holding 1 element must be implemented for the purpose of this communication.

1. **Software Design**
2. **Measurement Data**

****

|  |  |  |  |
| --- | --- | --- | --- |
| Task | With profiling | No profiling | W/ button push |
| PIDWork (10s) | 1432 | 1592 | 1519 |

|  |  |  |  |
| --- | --- | --- | --- |
| TIMESLICE (ms) | Data lost | Jitter (us) | PIDWork |
| 2 | 0 | 22 | 1004 |
| 1 | 0 | 22 | 1287 |
| 10 | 0 | 22 | 1262 |

1. **Analysis and Discussion**
2. If the UART is interrupt driven, than unpredictable high-priority interrupts will increase the jitter.
3. ADC data is collected via a hardware interrupt. Even if interrupts are disabled when the interrupt is ready, the data will still be stored into the register because int only disabled software interrupts and handlers.
4. DAS: priority 1; ADC: priority 3; Button: priority 2; SysTick: priority 7. The reason behind these priorities is to give the real-time tasks the highest priorities to minimize jitter.
5. If the stack size is too small, then if too large of a variable is allocated, or too many function calls are nested, the processor will try to write into a memory location that is not allocated specifically for the stack. To detect stack over/underflows, we can use the Memory Protection Unit. The Arm Cortex M3 includes an MPU which can specify regions in memory as protected and can be set to cause a fault on access these protected regions. In the fault handler, we can then kill the foreground thread which caused this fault.
6. The OS\_Kill() statements at the end of Consumer and Display should always be executed, as once the ADC has sampled enough data via Producer they should exit their while() loops and then execute OS\_Kill(). However, based on observation, this doesn't always happen.
7. *Deterministic* means that for a fixed input, the output is always the same. In this case, the context switch happens after 6 elements are put into the fifo. If the fifo has data, the consumer will get the data, and attempt to put it into thte mailbox. Data can also be lost if the mailbox is full and the consumer can’t get and send any more data.
8. Based on the length of the timeslice, it is unlikely that Consumer ever waits on OS\_MailBoxSend, as it has to wait for 64 samples from the ADC and then compute the FFT from them in between calls. As long as a thread switch happens during that time, allowing Display to empty the mailbox, it won't have to wait.

**Code:**

void OS\_InitSemaphore(OS\_SemaphoreType \*s, int permits) {

s->value = permits;

}

int OS\_AddThread(void(\*task)(void), unsigned long stackSize, unsigned long priority) {

OS\_CRITICAL\_FUNCTION;

\_TCB\* thread;// = (\_TCB\*) malloc(sizeof(\_TCB));

int i, tid = \_OS\_numThreads++;

// lock thread linked list

OS\_bWait(&\_modifyTCB);

OS\_ENTER\_CRITICAL();

if(\_RunPt == NULL) {

// this is the first thread

thread = &\_threads[0];

\_OS\_InitThread(thread, task, thread, thread, tid, 0, 0, priority) // TODO: deide how to handle stacks

\_RunPt = thread;

}

else {

// need to find the first open space in thread storage array

for(i = 0; i < \_OS\_MAX\_THREADS; i++) {

if(\_threads[i].id == \_OS\_FREE\_THREAD) {

break;

}

}

if(i == \_OS\_MAX\_THREADS) {

// maximum number of threads already used

// TODO - handle this more elgeantly

OS\_bSignal(&\_modifyTCB);

return 0;

}

else {

\_TCB\* temp = \_RunPt; // store \_RunPt in case thread switcher changes it while this is happening

thread = &\_threads[i];

// add thread between RunPt.previous and RunPt, at the end of the list

\_OS\_InitThread(thread, task, temp, temp->prev, tid, 0, 0, priority)

temp->prev->next = thread;

temp->prev = thread;

}

}

// unlock thread linked list

OS\_EXIT\_CRITICAL();

OS\_bSignal(&\_modifyTCB);

return 1;

}

void OS\_Init(void) {

int i;

DisableInterrupts();

SysCtlClockSet(SYSCTL\_SYSDIV\_4 | SYSCTL\_USE\_PLL | SYSCTL\_XTAL\_8MHZ | SYSCTL\_OSC\_MAIN);

/\* Initialize SysTick \*/

NVIC\_ST\_CTRL\_R = 0; // disable SysTick during setup

NVIC\_ST\_CURRENT\_R = 0; // any write to current clears it

NVIC\_SYS\_PRI3\_R =(NVIC\_SYS\_PRI3\_R&0x00FFFFFF)|0xE0000000; // priority 7

/\* Initialize PendSV \*/

NVIC\_INT\_CTRL\_R = 0; // disable PendSV during setup, may not be necessary, or my be vital ?!?!

NVIC\_SYS\_PRI3\_R =(NVIC\_SYS\_PRI3\_R&0xFF00FFFF)|0x00E00000; // lowest priority

/\* Initialize foreground linked list \*/

OS\_InitSemaphore(&\_modifyTCB, OS\_BINARY\_SEMAPHORE);

\_RunPt = NULL;

// intitalize array of threads to indicate they are all free

for(i = 0; i < \_OS\_MAX\_THREADS; i++) {

\_threads[i].id = \_OS\_FREE\_THREAD;

}

\_OS\_numThreads = 0;

// initialize timers used by OS

Timer2A\_Init();

Timer2B\_Init(2);

/\* Add default thread in case all threads killed \*/

OS\_AddThread(&\_OS\_Default\_Thread, 0, 0); // should be lowest priority

}

// return the id of the thread pointed to by \_RunPt

// if \_RunPt is null, return -1

int OS\_Id(void) {

if(\_RunPt == NULL) {

return -1;

}

return \_RunPt->id;

}

// redefine for debugging that needs access to private variables

void OS\_Debug(void) {

UART\_OutString("threads:\r\n");

\_OS\_PrintThreads(3);

}

// print information from n threads, starting with RunPt and cycling if necessary

static void \_OS\_PrintThreads(int n) {

int i;

\_TCB \*temp;

if (\_RunPt != NULL) {

temp = \_RunPt;

for(i = 0; i < n; i++) {

\_OS\_PrintThread(temp);

temp = temp->next;

}

}

else {

UART\_OutString("No threads"); UART\_OutString(SH\_NL);

}

}

static void \_OS\_PrintThread(\_TCB\* thread) {

UART\_OutString("this = "); UART\_OutUDec(thread->id); UART\_OutString(SH\_NL);

UART\_OutString("next = "); UART\_OutUDec(thread->next->id); UART\_OutString(SH\_NL);

UART\_OutString("prev = "); UART\_OutUDec(thread->prev->id); UART\_OutString(SH\_NL);

UART\_OutString("priority = "); UART\_OutUDec(thread->priority); UART\_OutString(SH\_NL);

}

static void \_OS\_SetInitialStack(\_TCB\* thread, void(\*task)(void)){

// thread->stack = (unsigned long\*) malloc(\_OS\_STACK\_SIZE \* sizeof(unsigned long));

thread->sp = &thread->stack[\_OS\_STACK\_SIZE-16]; // thread stack pointer

thread->stack[\_OS\_STACK\_SIZE-1] = 0x01000000; // thumb bit

thread->stack[\_OS\_STACK\_SIZE-2] = (unsigned long) task; // initial pc

thread->stack[\_OS\_STACK\_SIZE-3] = 0x14141414; // R14

thread->stack[\_OS\_STACK\_SIZE-4] = 0x12121212; // R12

thread->stack[\_OS\_STACK\_SIZE-5] = 0x03030303; // R3

thread->stack[\_OS\_STACK\_SIZE-6] = 0x02020202; // R2

thread->stack[\_OS\_STACK\_SIZE-7] = 0x01010101; // R1

thread->stack[\_OS\_STACK\_SIZE-8] = 0x00000000; // R0

thread->stack[\_OS\_STACK\_SIZE-9] = 0x11111111; // R11

thread->stack[\_OS\_STACK\_SIZE-10] = 0x10101010; // R10

thread->stack[\_OS\_STACK\_SIZE-11] = 0x09090909; // R9

thread->stack[\_OS\_STACK\_SIZE-12] = 0x08080808; // R8

thread->stack[\_OS\_STACK\_SIZE-13] = 0x07070707; // R7

thread->stack[\_OS\_STACK\_SIZE-14] = 0x06060606; // R6

thread->stack[\_OS\_STACK\_SIZE-15] = 0x05050505; // R5

thread->stack[\_OS\_STACK\_SIZE-16] = 0x04040404; // R4

}

void OS\_Launch(unsigned long theTimeSlice) {

NVIC\_ST\_RELOAD\_R = theTimeSlice - 1; // reload value

NVIC\_ST\_CTRL\_R = 0x00000007; // enable, core clock and interrupt arm

StartOS();

}

/\* Remove \_RunPt from the linked list \*/

void OS\_Kill(void)

{

int i;

\_TCB \*temp;

OS\_CRITICAL\_FUNCTION;

OS\_bWait(&\_modifyTCB);

OS\_ENTER\_CRITICAL();

\_RunPt->prev->next = \_RunPt->next;

\_RunPt->next->prev = \_RunPt->prev;

temp = \_RunPt;

\_RunPt = \_RunPt->next; // TODO - could a systick interrupt cause the next thread to be skipped?

// search tcb array to find it by checking unique id's

for(i = 0; i < \_OS\_MAX\_THREADS; i++) {

if(\_threads[i].id == temp->id) {

\_threads[i].id = \_OS\_FREE\_THREAD; // delete thread (symbolically)

break;

}

}

if(i == \_OS\_MAX\_THREADS) {

// TODO - this should never happen

}

OS\_EXIT\_CRITICAL();

OS\_bSignal(&\_modifyTCB);

NVIC\_ST\_CURRENT\_R = 0x0; // any write clears

NVIC\_INT\_CTRL\_R |= NVIC\_INT\_CTRL\_PEND\_SV; // trigger pendSV interrupt

OS\_Delay(OS\_ARBITRARY\_DELAY);

}

static void \_OS\_Default\_Thread(void) {

while(1)

OS\_Suspend();

}

void OS\_Suspend(void) {

NVIC\_ST\_CURRENT\_R = 0x0; // any write clears

NVIC\_INT\_CTRL\_R |= NVIC\_INT\_CTRL\_PENDSTSET; // trigger a systick interrupt to switch threads

OS\_Delay(OS\_ARBITRARY\_DELAY);

}

void OS\_Delay(int count) {

int i;

for(i = 0; i < count; i++)

;

}

int OS\_AddButtonTask(void(\*task)(void), unsigned long priority) {

static unsigned int haveInit = 0; // only initialize once

if(!haveInit) {

PORTF\_Init(); // initialize; for now, just select switch (PF1)

haveInit = 1;

}

// initialize NVIC interrupts for port F

NVIC\_PRI7\_R = ((NVIC\_PRI7\_R&0xFF0FFFFFFF)

| (priority << 21));

NVIC\_EN0\_R |= NVIC\_EN0\_INT30;

\_OS\_SelTask = task; // TODO - handle priority

return 1;

}

//static unsigned long LastPF1 = 1;

void GPIOPortF\_Handler(void) {

// if(LastPF1 == 1) {

if(\_OS\_SelTask != NULL) {

\_OS\_SelTask();

}

// }

GPIO\_PORTF\_IM\_R &= ~PORTF\_PINS; // disarm interrupt

OS\_AddThread(&DebounceTask, \_OS\_STACK\_SIZE, 5); // TODO - handle priority

}

static void DebounceTask(void) {

OS\_Sleep(200); // foreground sleeping, must run within 50ms

// LastPF1 = PF1; // read while it is not bouncing

GPIO\_PORTF\_ICR\_R |= PORTF\_PINS; // acknowledge interrupt

GPIO\_PORTF\_IM\_R |= PORTF\_PINS; // re-arm interrupt

OS\_Kill();

OS\_Delay(OS\_ARBITRARY\_DELAY);

}

void OS\_Sleep(unsigned long sleepTime) {

\_RunPt->sleepTime = sleepTime; // how long to sleep

\_RunPt->sleep = 1; // indicate this thread is now sleeping

OS\_Suspend(); // trigger thread switch

}

void OS\_FindNextThread(void) {

while(\_RunPt->sleep || \_RunPt->block) {

\_RunPt = \_RunPt->next;

}

}

void OS\_Fifo\_Init(unsigned long size) {

memset(\_OS\_Fifo.Fifo, 0, sizeof(\_OS\_Fifo.Fifo)); // initialize all values to 0

\_OS\_Fifo.PutIndex = \_OS\_Fifo.GetIndex = 0;

OS\_InitSemaphore(&\_OS\_Fifo.notEmpty, 0);

OS\_InitSemaphore(&\_OS\_Fifo.mutex, OS\_BINARY\_SEMAPHORE);

}

int OS\_Fifo\_Put(unsigned long data) {

// NOT THREAD SAFE!!

if((\_OS\_Fifo.PutIndex + 1) == \_OS\_Fifo.GetIndex) {

return 0;

}

\_OS\_Fifo.Fifo[\_OS\_Fifo.PutIndex] = data;

\_OS\_Fifo.PutIndex = (\_OS\_Fifo.PutIndex + 1) & (\_OS\_FIFO\_SIZE - 1);

OS\_Signal(&\_OS\_Fifo.notEmpty);

return 1;

}

unsigned long OS\_Fifo\_Get(void) {

unsigned long data;

OS\_Wait(&\_OS\_Fifo.notEmpty);

OS\_bWait(&\_OS\_Fifo.mutex);

data = \_OS\_Fifo.Fifo[\_OS\_Fifo.GetIndex];

\_OS\_Fifo.GetIndex = (\_OS\_Fifo.GetIndex + 1) & (\_OS\_FIFO\_SIZE - 1);

OS\_bSignal(&\_OS\_Fifo.mutex);

return data;

}

long OS\_Fifo\_Size(void) {

return (\_OS\_Fifo.PutIndex - \_OS\_Fifo.GetIndex) & (\_OS\_FIFO\_SIZE - 1);

}

void OS\_MailBox\_Init(void) {

\_OS\_Mailbox.data = 0;

OS\_InitSemaphore(&\_OS\_Mailbox.hasData, 0);

OS\_InitSemaphore(&\_OS\_Mailbox.gotData, 1);

}

void OS\_MailBox\_Send(unsigned long data) {

OS\_bWait(&\_OS\_Mailbox.gotData);

\_OS\_Mailbox.data = data;

OS\_bSignal(&\_OS\_Mailbox.hasData);

}

unsigned long OS\_MailBox\_Recv(void) {

unsigned long data;

OS\_bWait(&\_OS\_Mailbox.hasData);

data = \_OS\_Mailbox.data;

OS\_bSignal(&\_OS\_Mailbox.gotData);

return data;

}

unsigned long OS\_Time(void) {

return TIMER2\_TBR\_R / 50;//\_us10Count \* 10; // 1us = 20ns \* 50

}

unsigned long OS\_TimeDifference(unsigned long start, unsigned long stop) {

if(stop > start) {

return (stop - start); // inputs should already be in 20ns units

}

return start - stop;

}