// os.c

// Runs on LM4F120/TM4C123/MSP432

// A priority/blocking real-time operating system

// Lab 4 starter file.

// Daniel Valvano

// March 25, 2016

// Hint: Copy solutions from Lab 3 into Lab 4

// Completed by Guillaume Galasso with a score of 100%

#include <stdint.h>

#include "os.h"

#include "CortexM.h"

#include "BSP.h"

#include "../inc/tm4c123gh6pm.h"

// function definitions in osasm.s

void StartOS(void);

#define NUMTHREADS 8 // maximum number of threads

#define NUMPERIODIC 2 // maximum number of periodic threads

#define STACKSIZE 100 // number of 32-bit words in stack per thread

struct tcb{

int32\_t \*sp; // pointer to stack (valid for threads not running

struct tcb \*next; // linked-list pointer

int32\_t \*blocked; // nonzero if blocked on this semaphore

int32\_t sleep; // nonzero if this thread is sleeping

uint32\_t priority;

//\*FILL THIS IN\*\*\*\*

};

typedef struct tcb tcbType;

tcbType tcbs[NUMTHREADS+NUMPERIODIC];

tcbType \*RunPt;

int32\_t Stacks[NUMTHREADS+NUMPERIODIC][STACKSIZE];

void static runperiodicevents(void);

// \*\*\*\*\*\*\*\* OS\_Init \*\*\*\*\*\*\*\*\*\*\*\*

// Initialize operating system, disable interrupts

// Initialize OS controlled I/O: periodic interrupt, bus clock as fast as possible

// Initialize OS global variables

// Inputs: none

// Outputs: none

void OS\_Init(void){

DisableInterrupts();

BSP\_Clock\_InitFastest();// set processor clock to fastest speed

// perform any initializations needed

BSP\_PeriodicTask\_Init(&runperiodicevents,1000,6); //1 ms hardware interrupt to decrement sleep;

}

void SetInitialStack(int i){

tcbs[i].sp = &Stacks[i][STACKSIZE-16]; // thread stack pointer

Stacks[i][STACKSIZE-1] = 0x01000000; // Thumb bit

Stacks[i][STACKSIZE-3] = 0x14141414; // R14

Stacks[i][STACKSIZE-4] = 0x12121212; // R12

Stacks[i][STACKSIZE-5] = 0x03030303; // R3

Stacks[i][STACKSIZE-6] = 0x02020202; // R2

Stacks[i][STACKSIZE-7] = 0x01010101; // R1

Stacks[i][STACKSIZE-8] = 0x00000000; // R0

Stacks[i][STACKSIZE-9] = 0x11111111; // R11

Stacks[i][STACKSIZE-10] = 0x10101010; // R10

Stacks[i][STACKSIZE-11] = 0x09090909; // R9

Stacks[i][STACKSIZE-12] = 0x08080808; // R8

Stacks[i][STACKSIZE-13] = 0x07070707; // R7

Stacks[i][STACKSIZE-14] = 0x06060606; // R6

Stacks[i][STACKSIZE-15] = 0x05050505; // R5

Stacks[i][STACKSIZE-16] = 0x04040404; // R4

// \*\*Same as Lab 2 and 3\*\*\*\*

}

//\*\*\*\*\*\*\*\* OS\_AddThreads \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Add eight main threads to the scheduler

// Inputs: function pointers to eight void/void main threads

// priorites for each main thread (0 highest)

// Outputs: 1 if successful, 0 if this thread can not be added

// This function will only be called once, after OS\_Init and before OS\_Launch

int OS\_AddThreads(void(\*thread0)(void), uint32\_t p0,

void(\*thread1)(void), uint32\_t p1,

void(\*thread2)(void), uint32\_t p2,

void(\*thread3)(void), uint32\_t p3,

void(\*thread4)(void), uint32\_t p4,

void(\*thread5)(void), uint32\_t p5,

void(\*thread6)(void), uint32\_t p6,

void(\*thread7)(void), uint32\_t p7){

// \*\*similar to Lab 3. initialize priority field\*\*\*\*

int32\_t status;

status = StartCritical();

tcbs[0].next = &tcbs[1]; // 0 points to 1

tcbs[1].next = &tcbs[2]; // 1 points to 2

tcbs[2].next = &tcbs[3]; // 2 points to 3

tcbs[3].next = &tcbs[4]; // 3 points to 4

tcbs[4].next = &tcbs[5]; // 4 points to 5

tcbs[5].next = &tcbs[6]; // 5 points to 6

tcbs[6].next = &tcbs[7]; // 6 points to 7

tcbs[7].next = &tcbs[0]; // 7 points to 0

SetInitialStack(0); Stacks[0][STACKSIZE-2] = (int32\_t)(thread0); // PC to address of thread

tcbs[0].blocked = 0; tcbs[0].sleep = 0; tcbs[0].priority = p0;

SetInitialStack(1); Stacks[1][STACKSIZE-2] = (int32\_t)(thread1); // PC

tcbs[1].blocked = 0; tcbs[1].sleep = 0; tcbs[1].priority = p1;

SetInitialStack(2); Stacks[2][STACKSIZE-2] = (int32\_t)(thread2); // PC

tcbs[2].blocked = 0; tcbs[2].sleep = 0; tcbs[2].priority = p2;

SetInitialStack(3); Stacks[3][STACKSIZE-2] = (int32\_t)(thread3); // PC

tcbs[3].blocked = 0; tcbs[3].sleep = 0; tcbs[3].priority = p3;

SetInitialStack(4); Stacks[4][STACKSIZE-2] = (int32\_t)(thread4); // PC

tcbs[4].blocked = 0; tcbs[4].sleep = 0; tcbs[4].priority = p4;

SetInitialStack(5); Stacks[5][STACKSIZE-2] = (int32\_t)(thread5); // PC

tcbs[5].blocked = 0; tcbs[5].sleep = 0; tcbs[5].priority = p5;

SetInitialStack(6); Stacks[6][STACKSIZE-2] = (int32\_t)(thread6); // PC

tcbs[6].blocked = 0; tcbs[6].sleep = 0; tcbs[6].priority = p6;

SetInitialStack(7); Stacks[7][STACKSIZE-2] = (int32\_t)(thread7); // PC

tcbs[7].blocked = 0; tcbs[7].sleep = 0; tcbs[7].priority = p7;

RunPt = &tcbs[0]; // thread 0 will run first

EndCritical(status);

return 1; // successful

}

void static runperiodicevents(void){

// \*\*\*\*IMPLEMENT THIS\*\*\*\*

// \*\*DECREMENT SLEEP COUNTERS

// In Lab 4, handle periodic events in RealTimeEvents

for (int i=0; i<NUMTHREADS; i++){

if (tcbs[i].sleep) {

tcbs[i].sleep--;

}

}

}

//\*\*\*\*\*\*\*\* OS\_Launch \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Start the scheduler, enable interrupts

// Inputs: number of clock cycles for each time slice

// Outputs: none (does not return)

// Errors: theTimeSlice must be less than 16,777,216

void OS\_Launch(uint32\_t theTimeSlice){

STCTRL = 0; // disable SysTick during setup

STCURRENT = 0; // any write to current clears it

SYSPRI3 =(SYSPRI3&0x00FFFFFF)|0xE0000000; // priority 7

STRELOAD = theTimeSlice - 1; // reload value

STCTRL = 0x00000007; // enable, core clock and interrupt arm

StartOS(); // start on the first task

}

// runs every ms

void Scheduler(void){ // every time slice

// \*\*\*\*IMPLEMENT THIS\*\*\*\*

// look at all threads in TCB list choose

// highest priority thread not blocked and not sleeping

// If there are multiple highest priority (not blocked, not sleeping) run these round robin

uint32\_t max = 255; // max

tcbType \*pt;

tcbType \*bestPt;

pt = RunPt; // search for highest thread not blocked or sleeping

do{

pt = pt->next; // skips at least one

if((pt->priority < max)&&((pt->blocked)==0)&&((pt->sleep)==0)){

max = pt->priority;

bestPt = pt;

}

} while(RunPt != pt); // look at all possible threads

RunPt = bestPt;

}

//\*\*\*\*\*\*\*\* OS\_Suspend \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Called by main thread to cooperatively suspend operation

// Inputs: none

// Outputs: none

// Will be run again depending on sleep/block status

void OS\_Suspend(void){

STCURRENT = 0; // any write to current clears it

INTCTRL = 0x04000000; // trigger SysTick

// next thread gets a full time slice

}

// \*\*\*\*\*\*\*\* OS\_Sleep \*\*\*\*\*\*\*\*\*\*\*\*

// place this thread into a dormant state

// input: number of msec to sleep

// output: none

// OS\_Sleep(0) implements cooperative multitasking

void OS\_Sleep(uint32\_t sleepTime){

// \*\*\*\*IMPLEMENT THIS\*\*\*\*

// set sleep parameter in TCB, same as Lab 3

// suspend, stops running

RunPt->sleep = sleepTime;

OS\_Suspend();

}

// \*\*\*\*\*\*\*\* OS\_InitSemaphore \*\*\*\*\*\*\*\*\*\*\*\*

// Initialize counting semaphore

// Inputs: pointer to a semaphore

// initial value of semaphore

// Outputs: none

void OS\_InitSemaphore(int32\_t \*semaPt, int32\_t value){

// \*\*\*\*IMPLEMENT THIS\*\*\*\*

// Same as Lab 3

\*semaPt = value;

}

// \*\*\*\*\*\*\*\* OS\_Wait \*\*\*\*\*\*\*\*\*\*\*\*

// Decrement semaphore and block if less than zero

// Lab2 spinlock (does not suspend while spinning)

// Lab3 block if less than zero

// Inputs: pointer to a counting semaphore

// Outputs: none

void OS\_Wait(int32\_t \*semaPt){

// \*\*\*\*IMPLEMENT THIS\*\*\*\*

// Same as Lab 3

DisableInterrupts();

\*semaPt = \*semaPt - 1;

if((\*semaPt) < 0) {

RunPt->blocked = semaPt; // reason it is blocked

EnableInterrupts();

OS\_Suspend(); // run thread switcher

}

EnableInterrupts();

}

// \*\*\*\*\*\*\*\* OS\_Signal \*\*\*\*\*\*\*\*\*\*\*\*

// Increment semaphore

// Lab2 spinlock

// Lab3 wakeup blocked thread if appropriate

// Inputs: pointer to a counting semaphore

// Outputs: none

void OS\_Signal(int32\_t \*semaPt){

// \*\*\*\*IMPLEMENT THIS\*\*\*\*

// Same as Lab 3

tcbType \*pt;

DisableInterrupts();

\*semaPt = \*semaPt + 1;

if((\*semaPt) <= 0){

pt = RunPt->next;

while(pt->blocked != semaPt){

pt = pt->next; // search for a thread blocked on this semaphore

}

pt->blocked = 0; // wakeup this one

}

EnableInterrupts();

}

#define FSIZE 10 // can be any size

uint32\_t PutI; // index of where to put next

uint32\_t GetI; // index of where to get next

uint32\_t Fifo[FSIZE];

int32\_t CurrentSize;// 0 means FIFO empty, FSIZE means full

uint32\_t LostData; // number of lost pieces of data

// \*\*\*\*\*\*\*\* OS\_FIFO\_Init \*\*\*\*\*\*\*\*\*\*\*\*

// Initialize FIFO. The "put" and "get" indices initially

// are equal, which means that the FIFO is empty. Also

// initialize semaphores to track properties of the FIFO

// such as size and busy status for Put and Get operations,

// which is important if there are multiple data producers

// or multiple data consumers.

// Inputs: none

// Outputs: none

void OS\_FIFO\_Init(void){

// \*\*\*\*IMPLEMENT THIS\*\*\*\*

// Same as Lab 3

PutI = GetI = 0; // empty

OS\_InitSemaphore(&CurrentSize, 0);

LostData = 0;

}

// \*\*\*\*\*\*\*\* OS\_FIFO\_Put \*\*\*\*\*\*\*\*\*\*\*\*

// Put an entry in the FIFO. Consider using a unique

// semaphore to wait on busy status if more than one thread

// is putting data into the FIFO and there is a chance that

// this function may interrupt itself.

// Inputs: data to be stored

// Outputs: 0 if successful, -1 if the FIFO is full

int OS\_FIFO\_Put(uint32\_t data){

// \*\*\*\*IMPLEMENT THIS\*\*\*\*

// Same as Lab 3

if (CurrentSize == FSIZE) {

LostData++;

return -1; //full

} else {

Fifo[PutI] = data; // put

PutI = (PutI+1)%FSIZE; //wrap index

OS\_Signal(&CurrentSize);

return 0; //success

}

}

// \*\*\*\*\*\*\*\* OS\_FIFO\_Get \*\*\*\*\*\*\*\*\*\*\*\*

// Get an entry from the FIFO. Consider using a unique

// semaphore to wait on busy status if more than one thread

// is getting data from the FIFO and there is a chance that

// this function may interrupt itself.

// Inputs: none

// Outputs: data retrieved

uint32\_t OS\_FIFO\_Get(void){uint32\_t data;

// \*\*\*\*IMPLEMENT THIS\*\*\*\*

// Same as Lab 3

OS\_Wait(&CurrentSize); // block if empty

data = Fifo[GetI];

GetI = (GetI+1)%FSIZE; //wrap index

return data;

}

// \*\*\*\*\*periodic events\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int32\_t \*PeriodicSemaphore0;

uint32\_t Period0; // time between signals

int32\_t \*PeriodicSemaphore1;

uint32\_t Period1; // time between signals

void RealTimeEvents(void){int flag=0;

static int32\_t realCount = -10; // let all the threads execute once

// Note to students: we had to let the system run for a time so all user threads ran at least one

// before signalling the periodic tasks

realCount++;

if(realCount >= 0){

if((realCount%Period0)==0){

OS\_Signal(PeriodicSemaphore0);

flag = 1;

}

if((realCount%Period1)==0){

OS\_Signal(PeriodicSemaphore1);

flag=1;

}

if(flag){

OS\_Suspend();

}

}

}

// \*\*\*\*\*\*\*\* OS\_PeriodTrigger0\_Init \*\*\*\*\*\*\*\*\*\*\*\*

// Initialize periodic timer interrupt to signal

// Inputs: semaphore to signal

// period in ms

// priority level at 0 (highest

// Outputs: none

void OS\_PeriodTrigger0\_Init(int32\_t \*semaPt, uint32\_t period){

PeriodicSemaphore0 = semaPt;

Period0 = period;

BSP\_PeriodicTask\_InitC(&RealTimeEvents,1000,0);

}

// \*\*\*\*\*\*\*\* OS\_PeriodTrigger1\_Init \*\*\*\*\*\*\*\*\*\*\*\*

// Initialize periodic timer interrupt to signal

// Inputs: semaphore to signal

// period in ms

// priority level at 0 (highest

// Outputs: none

void OS\_PeriodTrigger1\_Init(int32\_t \*semaPt, uint32\_t period){

PeriodicSemaphore1 = semaPt;

Period1 = period;

BSP\_PeriodicTask\_InitC(&RealTimeEvents,1000,0);

}

//\*\*\*\*edge-triggered event\*\*\*\*\*\*\*\*\*\*\*\*

int32\_t \*edgeSemaphore;

// \*\*\*\*\*\*\*\* OS\_EdgeTrigger\_Init \*\*\*\*\*\*\*\*\*\*\*\*

// Initialize button1, PD6, to signal on a falling edge interrupt

// Inputs: semaphore to signal

// priority

// Outputs: none

void OS\_EdgeTrigger\_Init(int32\_t \*semaPt, uint8\_t priority){

edgeSemaphore = semaPt;

//\*\*\*IMPLEMENT THIS\*\*\*

SYSCTL\_RCGCGPIO\_R |= 0x00000008; // 1) activate clock for Port D

while((SYSCTL\_PRGPIO\_R&0x08) == 0){}; // allow time for clock to stabilize

GPIO\_PORTD\_AMSEL\_R &= ~0x40; // 2) disable analog on PD6

GPIO\_PORTD\_PCTL\_R = (GPIO\_PORTD\_PCTL\_R&0xF0FFFFFF)+0x00000000; // 3) configure PD6 as GPIO

GPIO\_PORTD\_DIR\_R &= ~0x40; // 4) make PD6 input

GPIO\_PORTD\_AFSEL\_R &= ~0x40;// 5) disable alt funct on PD6

GPIO\_PORTD\_PUR\_R &= ~0x40; // disable pull-up on PD6

GPIO\_PORTD\_DEN\_R |= 0x40; // 6) enable digital I/O on PD6

GPIO\_PORTD\_IS\_R &= ~0x40; // PD6 is edge-sensitive

GPIO\_PORTD\_IBE\_R &= ~0x40; // PD6 is not both edges

GPIO\_PORTD\_IEV\_R &= ~0x40; // PD6 is falling edge event

GPIO\_PORTD\_ICR\_R = 0x40; // clear PD6 flag

GPIO\_PORTD\_IM\_R |= 0x40; // arm interrupt on PD6

NVIC\_PRI0\_R = (NVIC\_PRI0\_R&0x1FFFFFFF)|((uint32\_t)(priority) << 29); // priority on Port D edge trigger is NVIC\_PRI0\_R 31 ? 29

NVIC\_EN0\_R = 0x00000008; // enable is bit 3 in NVIC\_EN0\_R

}

// \*\*\*\*\*\*\*\* OS\_EdgeTrigger\_Restart \*\*\*\*\*\*\*\*\*\*\*\*

// restart button1 to signal on a falling edge interrupt

// rearm interrupt

// Inputs: none

// Outputs: none

void OS\_EdgeTrigger\_Restart(void){

//\*\*\*IMPLEMENT THIS\*\*\*

GPIO\_PORTD\_IM\_R |= 0x40; // rearm interrupt 3 in NVIC

GPIO\_PORTD\_ICR\_R = 0x40; // clear flag6

}

void GPIOPortD\_Handler(void){

//\*\*\*IMPLEMENT THIS\*\*\*

GPIO\_PORTD\_ICR\_R = 0x40; // step 1 acknowledge by clearing flag

OS\_Signal(edgeSemaphore); // step 2 signal semaphore (no need to run scheduler)

GPIO\_PORTD\_IM\_R &= ~0x40;// step 3 disarm interrupt to prevent bouncing to create multiple signals

OS\_Suspend();

}