/\*

\* kernel.c

\*

\* Created: 4/17/2020 3:10:47 PM

\* Author: Nathan Potvin

\*/

#include <avr/io.h>

#include <util/atomic.h>

#include "kernel.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* External function declarations

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

extern void \_\_attribute\_\_ ((naked)) schedule();

extern void init\_system\_timer();

extern void init\_serial();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Local function declarations

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void push\_pthread(uint8\_t tid, PTHREAD func);

void copy\_stack(uint8\_t \*, uint8\_t \*, volatile uint8\_t \*\*);

// macro to initialize a thread's control structure and stack canary

#define THREAD\_INIT(tid, stack, stack\_size) \

kernel\_data.thread\_ctrl\_tbl[tid].stack\_ptr = stack + stack\_size - 1; \

kernel\_data.thread\_ctrl\_tbl[tid].stack\_base = stack + stack\_size - 1;\

kernel\_data.thread\_ctrl\_tbl[tid].canary\_ptr = stack; \

\*(kernel\_data.thread\_ctrl\_tbl[tid].canary\_ptr) = CANARY; \

kernel\_data.thread\_ctrl\_tbl[tid].entry\_pnt \

= (PTHREAD) uninitialized\_thread\_error;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Kernel function definitions

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

\* Initialized the kernel and copies the stack to thread0's stack and returns

\* executing in thread0

\*/

void init()

{

// initialize kernel data and enable interrupts on exit

*ATOMIC\_BLOCK*(*ATOMIC\_FORCEON*)

{

// initialize the thread control structure and stack canary for

// each thread

THREAD\_INIT(THREAD0, kernel\_data.stacks.stack0, T0\_STACKSZ);

THREAD\_INIT(THREAD1, kernel\_data.stacks.stack1, T1\_STACKSZ);

THREAD\_INIT(THREAD2, kernel\_data.stacks.stack2, T2\_STACKSZ);

THREAD\_INIT(THREAD3, kernel\_data.stacks.stack3, T3\_STACKSZ);

THREAD\_INIT(THREAD4, kernel\_data.stacks.stack4, T4\_STACKSZ);

THREAD\_INIT(THREAD5, kernel\_data.stacks.stack5, T5\_STACKSZ);

THREAD\_INIT(THREAD6, kernel\_data.stacks.stack6, T6\_STACKSZ);

THREAD\_INIT(THREAD7, kernel\_data.stacks.stack7, T7\_STACKSZ);

// copy the stack to the thread 0 stack and set the stack pointer

// register to thread0's stack pointer

copy\_stack(GCC\_STACK\_BASE, \*(uint8\_t \*\*)STACK\_POINTER

, &(kernel\_data.thread\_ctrl\_tbl[THREAD0].stack\_ptr));

\*STACK\_POINTER = kernel\_data.thread\_ctrl\_tbl[THREAD0].stack\_ptr;

// initialize the disable status to disable all but thread 0

kernel\_data.schedule\_ctrl.disable\_status =

THREAD1\_MSK | THREAD2\_MSK | THREAD3\_MSK | THREAD4\_MSK

| THREAD5\_MSK | THREAD6\_MSK | THREAD7\_MSK;

// initialize the delay\_status so no threads are delayed

kernel\_data.schedule\_ctrl.delay\_status = 0x00;

// initialize the current thread and current thread mask to thread0

kernel\_data.schedule\_ctrl.cur\_thread\_id = THREAD0;

kernel\_data.schedule\_ctrl.cur\_thread\_msk = THREAD0\_MSK;

// initialize other functionality

init\_system\_timer();

# ifdef SERIAL

init\_serial();

# endif /\* SERIAL \*/

}

}

/\*

\* Initializes a thread's stack to begin execution at a given entry point.

\* If the given tid is the same as the calling thread, the scheduler will

\* be invoked and new will not return

\*

\* tid: thread id of the thread being initialized

\* entry\_point: function pointer that will be the thread's entry point.

\* Must be a no arg function. Any functions above 0xffff

\* in program memory must make use of a trampoline table

\* enabled: boolean value. True if the thread should be enabled

\* when this function exits

\*/

void new(uint8\_t tid, PTHREAD entry\_point, bool enabled)

{

*ATOMIC\_BLOCK*(*ATOMIC\_RESTORESTATE*)

{

kernel\_data.thread\_ctrl\_tbl[tid].stack\_ptr =

kernel\_data.thread\_ctrl\_tbl[tid].stack\_base;

kernel\_data.thread\_ctrl\_tbl[tid].entry\_pnt = entry\_point;

push\_pthread(tid, entry\_point);

kernel\_data.thread\_ctrl\_tbl[tid].stack\_ptr -= THREAD\_STACK\_CONTEXT\_SZ;

// for preemptive builds initialize the status register

// part of the context to have the interrupt enabled

# ifdef PREEMPTIVE

\*(kernel\_data.thread\_ctrl\_tbl[tid].stack\_ptr + 1) = 0x80;

# endif /\* PREEMPTIVE \*/

if (enabled)

{

kernel\_data.schedule\_ctrl.disable\_status &= ~(1<<tid);

}

else

{

kernel\_data.schedule\_ctrl.disable\_status |= 1<<tid;

}

if (kernel\_data.schedule\_ctrl.cur\_thread\_id == tid)

{

schedule();

}

}

}

/\*

\* enabled the specified thread allowing it to be scheduled

\*

\* tid: thread id of the thread to be enabled

\*/

void enable(uint8\_t tid)

{

*ATOMIC\_BLOCK*(*ATOMIC\_RESTORESTATE*)

{

kernel\_data.schedule\_ctrl.disable\_status &= ~(1<<tid);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Local function definitions

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

\* Pushes a given function pointer to the stack of a given thread.

\* The function pointer is pushed big endian to allow it to be returned

\* to with the ret instruction

\*

\* tid: the thread id of the stack that the function pointer will be

\* pushed too

\* func: the function pointer to be pushed

\*/

void push\_pthread(uint8\_t tid, PTHREAD func)

{

// push low order byte

\*(kernel\_data.thread\_ctrl\_tbl[tid].stack\_ptr--) =

(uint8\_t) ((((*uint16\_t*) func) & 0x00ff) >> 0);

// push middle order byte

\*(kernel\_data.thread\_ctrl\_tbl[tid].stack\_ptr--) =

(uint8\_t) ((((*uint16\_t*) func) & 0xff00) >> 8);

// push high order byte

// because the ATmega2560 has a 17 bit address space for program memory

// and gcc uses 16 bit address space, the high order byte is always zero

// for a function pointer and trampoline tables are used for functions

// in high parts of program memory

\*(kernel\_data.thread\_ctrl\_tbl[tid].stack\_ptr--) = (uint8\_t) (0x00);

}

/\*

\* Copies the stack defined by s1\_base and s1\_ptr to the stack defined by

\* s2\_ptr

\*

\* s1\_base: pointer to the base of the stack to be copied

\* s1\_ptr: the stack pointer of the stack to be copied

\* s2\_ptr: a pointer to the stack pointer of the stack to be copied to

\*/

void copy\_stack(uint8\_t \*s1\_base, uint8\_t \*s1\_ptr, volatile uint8\_t \*\*s2\_ptr)

{

while (s1\_base > s1\_ptr)

{

\*\*s2\_ptr = \*s1\_base;

--s1\_base;

--(\*s2\_ptr);

}

}