Dalhousie University
Department of Electrical and Computer Engineering

ECED 4402 – Real Time Systems

Assignment 2: A light-weight messaging kernel

1 Objectives

In this assignment you are to write a light-weight kernel that allows a number of different processes to run “simultaneously” by swapping them in a round-robin fashion.

There are to be three priority levels—processes with the higher priority are to run before processes at a lower priority. Priority levels are fixed and cannot change. A program developer should be allowed to initiate any number of processes (limited only by the amount of available process control block memory and stack memory). Once all processes have been initiated, the processes should be allowed to run and, when required, terminate.

Processes can operate as clients, servers, or both. The kernel is to support an Inter-Process Communication (IPC) system, allowing processes to send and receive messages. Messages are sent to queues rather than processes, requiring processes to bind to queues. Processes waiting for messages are to block until a message arrives.

A number of questions are raised in this assignment. You are not expected to answer them; however, keep in mind that they could make interested quiz or examination questions.

2 The Kernel

The kernel consists of device drivers for the clock interrupt handler (to control the time quantum of each process), the UART handler (to support terminal I/O), the list of processes and their associated tables, and support routines (to handle context switches, message passing, and terminating processes). Both the clock and UART handlers are to be taken from Assignment 1.

2.1 Kernel Data Structures

The main kernel data structure is a queue of pending processes (i.e., the *waiting-to-run* queue). The head of the queue can be the currently running process, while the tail of the queue can be the most recently run process. There are to be three queues, one for each priority (see below for discussion).

An entry on the waiting-to-run queue consists of the registers associated with the process and any other information unique to the process.

2.2 Processes

Each process in this assignment is a C function of type void. The code and data segments are common to all processes (*Why?*); however, each process is to be assigned its own stack segment (using *malloc()*). Unless deliberately shared, all global variables should be treated as constants (*Why?*).

Since the assignment does not require an explicit loader, all processes are to be registered before the kernel is started; accordingly, it will be necessary to design and implement a process registration function such as:

*int reg_proc(void (*func_name)(), unsigned pid, unsigned priority);*

reg_proc() takes the name of a function, *func_name*, a unique process identifier, *pid*, and a priority (1 through 4), *priority*; if the process can be created, TRUE is returned, otherwise FALSE.

reg_proc() is responsible for creating a unique stack for the process (populated with initial register values) as well as a process control block or PCB (consisting of the process's identifier, stack pointer, and links to the adjacent processes in the waiting-to-run queue or a blocked queue). Both the stack and the PCB should be obtained from dynamic memory (the sizes of the memory blocks supplied by *malloc()*).

Once all processes are registered, *reg_proc()* should not be called again.

2.2.1 Process Identifiers

Each process is assigned a unique process identifier, which is available to the process by calling the function *getpid()*:

pid = getpid();

The process's identifier has a variety of possible uses, such as being the value of the row on which the process is to write its data; if each id is unique, a process can indicate that it is running by writing to its row. The *getpid()* function must be handled by the kernel. Neither the PCB nor any other kernel structures are allowed to be accessed by processes; *getpid()* is accessed by a supervisor call.

2.2.2 Process Life Cycle

A process begins its "life" when *reg_proc()* is called and its stack is created; the initial stack must contain the initial register set (i.e., xPSR, PC, LR, R12, R3, R2, R1, and R0), with the PC holding the value of the function to be executed and LR containing the address of a function that calls a kernel termination subroutine to remove the process from its queue and return any memory allocated to the process. The kernel termination subroutine is accessed by a supervisor call.

Space for registers R4 through R11 should also be reserved on the stack; these registers must be saved whenever the kernel is called or an exception occurs.

2.2.3 Applications

Applications are processes. See section 4.

2.2.4 Start Up

When starting up the program, it will be necessary to perform any memory initialization first, followed by the registration of the processes. Once all of the processes have been registered, execution of the first process can begin. The startup code (using the MSP stack) is to initialize the PSP to the stack of the first process and then perform an EXC_RETURN with LR indicating thread mode and the PSP stack.

The first process should be the highest priority process, although this need not be the case (*Why?*).

2.2.5 Process Termination

When a process terminates, it should be removed from its waiting-to-run queue so that it will no longer execute. This raises a number of issues (these questions do not have to be answered but solutions to them are necessary):

- *How does a process "terminate"?*

  A process terminates when it leaves its procedure (by a return or by reaching the closing brace '}'). Since the process must "return" somewhere, it is suggested that it pass control a termination function. Rather than expecting the terminating procedure to explicitly call the termination procedure, the address of the termination procedure can be the value of LR on the process's initial stack, meaning that the final return passes control to the termination procedure. The termination procedure should then call the kernel (via a supervisor call) to have the process removed from the list of processes; this must be done inside the kernel (*Why?*). It is only possible for the currently running process to terminate (*Why?*).

- *What happens when there are no more processes to run?*

  In certain situations, there may not be any processes available for the kernel to allow to execute (all may have terminated or eventually, all may be blocked or waiting for I/O or an event to occur). In these cases, the machine must still "do something", since it can't "do nothing".

  What many systems implement is an *idle process* that runs when there is nothing else to run. The idle process is registered as any other process; however, it should *only* run when there is nothing else left to run (*Why?*).

  Hmmm. Perhaps the idle process can have its own priority queue?

2.3 Other Issues

In Thumb 2 mode, the Cortex saves a subset of its CPU registers on the running process's stack when a trap (i.e., a supervisor call, SVC) or interrupt (from SysTick or the UART) occurs, the registers are stored on the stack in the following order (the stack pointer is the address of the lowest memory location, holding the value of R0); see Figure 1.

Low memory	unsigned long R0	**Stack pointer**
	unsigned long R1	
	unsigned long R2	
	unsigned long R3	
	unsigned long R12	
	unsigned long LR	
	unsigned long PC	
High memory	unsigned long xPSR	

Figure 1: Stack contents after an exception or interrupt

Whenever an exception or interrupt occurs,[1] the above registers are saved on the *running* process's stack (the address specified by the process stack point or PSP). The Cortex then changes to the kernel's stack (the address determined by the main stack pointer or MSP), pushing the return address onto the stack; since control is to return to the code that caused the exception or was interrupted, the return address indicates that it occurred inside either the kernel (EXC_RETURN value 0xFFFFFFF9) or a process (EXC_RETURN value 0xFFFFFFFD). When leaving the kernel or interrupt handler, the stack used is determined by the EXC_RETURN value stored in the PC; this causes the Cortex to access either the kernel stack (MSP) or process stack (PSP) and remove the stacked registers (Figure 1) to restore the state of the machine to what it was prior to the exception or interrupt.

The Cortex has multiple interrupt levels, which means that a process can be interrupted by an ISR, which, in turn, can be interrupted by SysTick. Similarly, a supervisor call (SVC) can be interrupted by an ISR or SysTick, or both. Since either an SVC or an ISR can access the process queues, it is ill-advised to have SysTick perform the actual context switch should an SVC or ISR be updating it as well. To avoid having these potential race conditions, the Cortex supports an additional interrupt vector referred to as PendSV which can be accessed after SysTick, any ISRs, or SVC have been handled.

PendSV (or pending supervisor call) can be used to implement the actual context switch after all other kernel calls (i.e., interrupts and exceptions) have been handled. This means that the context switch is not performed by SysTick but by the code associated with the PendSV interrupt vector.

Kernel functions (such as SysTick or the messaging system) can request that the PendSV handler be called using the following (this should be done only if a context switch is required):

```
#define NVIC_INT_CTRL_R  (*((volatile unsigned long *) 0xE000ED04))
#define TRIGGER_PENDSV  0x10000000

void SomeKernelFunction(void)
{
/* Code specific to the kernel function */
…
/* Signal that the PendSV handler is to be called on exit */
NVIC_INT_CTRL_R |= TRIGGER_PENDSV;
}
```

This will result in the PendSV handler being called by the CPU as an exception. To ensure that the handler is executed last and that all other handlers are executed before it, the Cortex must be programmed so that PendSV handler is operating at priority 7 (the lowest, see page 179 in the Tiva Data Sheet); this is done during kernel initialization:[2]

```
#define NVIC_SYS_PRI3_R   (*((volatile unsigned long *) 0xE000ED20))
```

[1] By definition, interrupts are external to the machine, while exceptions are internal and normally classified into faults (CPU or hardware error) or trap (a software "interrupt"). Note that SysTick is considered an exception because it is part of the processor as opposed to the Tiva microcontroller.

[2] The *NVIC_SYS_PRI3_R* register also allows the priority of SysTick to be specified (page 179).

```
#define PENDSV_LOWEST_PRIORITY       0x00E00000
```

```
void Kernel_Initialization()
{
…
NVIC_SYS_PRI3_R |= PENDSV_LOWEST_PRIORITY;
}
```

The low-priority *PendSV_Handler()* code in this example performs the context switch. Note that because it is interruptible, interrupts from higher-priority devices are not blocked (as they would be if the swap code was performed by SysTick):

```
void PendSV_Handler()
{
/* Save running process */
save_registers();        /* Save active CPU registers in PCB */
next_process();
restore_registers();     /* Restore process's registers */
}
```

(Since the PendSV handler can be interrupted, it is advisable to disable interrupts when modifying the process queue.)

The function *next_process()* can be written as follows:

```
void next_process()
{
/* Save current SP */
running -> sp = get_PSP();
/* Change PSP to next process stack */
running = running -> next;
/* Start next process */
set_PSP(running -> sp);
}
```

The variable `running` is a pointer to a PCB which holds the process's stack pointer (R13, saved as the PSP) and the links to any adjacent PCBs:

```
struct pcb
{
unsigned long sp;       /* Stack pointer - r13 (PSP) */
struct pcb *next, *prev;          /* Links to adjacent PCBs */
};
```

The four functions *save_registers()*, *restore_registers()*, *get_PSP()*, and *set_PSP()* are supplied on the website in *process.c* as well as some of the data structures described here are found in process.h. The *save_registers()* and *restore_registers()* functions both access the current process stack pointed to by the PSP register.

Note that the *save_registers()* function might not be necessary in the above function if the first kernel function to be entered (for example, an SVC) saved the running process's registers beforehand. This will require careful design.

Since the CCS C compiler can use some of the unsaved registers R4 through R11 (for example, when using pointers), it is necessary to save and restore these registers in the called exception or interrupt handler before any kernel-specific code is executed (this holds true for *all* handlers, including those for SysTick and UART).

Control can be passed voluntarily to the kernel by the running process executing a supervisor call or SVC. An SVC is implemented as an exception and can be associated with a distinguishing code number or register arguments, or both. In this assignment, a number of kernel calls will be required, notably the startup, process termination, getid(), and the messaging functions. In order to distinguish between the different calls, it is necessary to probe into the process stack to obtain any arguments to the kernel.

Note, if passing arguments by the stack, it is necessary to declare stacked variables as volatile in order to ensure that the compiler doesn't optimize things away.

Finally, the instructions used to start the first process running are unique in that it is necessary to force the machine to start execution from the process stack rather than the default main stack; this is done by assigning 0xFFFFFFFD to LR inside an exception handler:

```
__asm(" movw LR,#0xFFFD");        /* Lower 16 [and clear top 16] */
__asm(" movt LR,#0xFFFF");        /* Upper 16 only */
__asm(" bx LR");                  /* Force return to PSP */
```

Thereafter, all exceptions or interrupts cause the machine to generate an EXC_RETURN value (LR) of 0xFFFFFFFD**,** ensuring that the return uses the values on the current process stack pointed to by PSP. Note there are two underscores ('_') prefixing the *asm()* macro.

3 Messages

Processes exchange data via *messages*. A message is a contiguous block of memory with a known address and size.

3.1 Transmitting Messages

An ISR or a process can send a message to a message queue.[3] The message is put onto a queue (typically a fixed-length queue associated with a process using the *bind()* command, discussed below) associated with a target process. Since modifying the queue should be treated as an *atomic* action, it should take place uninterrupted, inside the kernel; for example:[4]

```
void k_send_message(unsigned mq_id, void *msg, unsigned size)
{
/*
```

[3] Message queues are distinct from process queues (one holds messages, the other, processes). However, the queues can interact in that a processes waiting for a message from an (empty) message queue is blocked and put onto a process queue for blocked processes.

[4] The '*k*' prefix denotes a kernel function and a '*p*' prefix denotes a process function.

```
  Update the message queue for the process associated with ‘mq_id’
  with the address of the ‘msg’.  The number of bytes in the message is
  specified in ‘size’.  This is uninterruptible, called within the kernel.
*/
...
}
```

An ISR should call this function whenever it is to send a message to a process (via a message queue). If the process is in the running state, the message should be stored in the specified message queue; however, if the process is blocked on this message queue, the message should be made available to the process and the process placed onto the running queue (see below).

When a process sends a message, it is necessary to have a *context switch* from the process to the kernel, supplying the kernel with the necessary values (in this case, the queue identifier of the target message queue, the message or its address, and the message size):

```
void p_send_message(unsigned mq_id, void *msg, unsigned size)
{
/*
  Update the message queue for the process associated with ‘mq_id’
  with the address of the message, ‘msg’.  ‘Size’ indicates the number of
  bytes in the message.  This is interruptible.
*/
struct p_msg_struct pmsg;
pmsg . mqid = mq_id;
pmsg . msg = msg;
pmsg . sz = size;
pkcall(SEND_MESSAGE, &pmsg);
...
}
```

Once the message has been created, its address should be passed to a common process-kernel call function that traps to the kernel (via *SVC()*) with a command code (in this example, *SEND_MESSAGE*) and a pointer to the message structure. Inside the kernel, the message queue number, the pointer to the message, and the size of the message can all be passed to *k_send_message()*.

3.2 Receiving Messages

For a process to receive a message, it must call the kernel, which will return a message if it is available or when it arrives. The message queue in question must be specified; for example:

```
void p_get_message(unsigned mq_id, void *msg, unsigned size)
{
/* Return the message when it arrives */
struct p_msg_struct pmsg;
pmsg . mqid = mq_id;
pmsg . msg = msg;
pmsg . sz = size;
pkcall(GET_MESSAGE, &pmsg);
/* A message will be available at this point */
}
```

The function *pkcall()* calls the kernel via *SVC()* and the kernel function to retrieve messages (*k_get_message()*) is called. When this call is made, either there will be a message on the message queue or there won't. If there is a message on the queue, it can be returned immediately to the process

```
void k_get_message(unsigned mq_id, void *msg, unsigned size)
{
/*
  If a message exists, it is returned to the calling process.  Note that mq_id
  should be checked for validity
*/
if (msg_q[mq_id] . list != NULL)
{
   /* copy size bytes from message queue to location specified by msg */
}
}
```

However, if the message queue is empty, the process must be blocked until a message arrives on the queue. When the message does arrive, the message information (i.e., address of the message and the number of bytes) is available to the kernel (it is on the process stack).

In this case, the kernel send routine can look something like (note the limit on the number of bytes being copied):

```
void k_send_message(unsigned mq_id, void *msg, unsigned size)
{
PCB *pcb_ptr;
unsigned psize;
void *msg_ptr;
…
pcb_ptr = msg_q[mq_id] . owner;
...
if (pcb_ptr -> state == BLOCKED)
{
   /* copy bytes from msg to pcb -> msg_ptr */
   /* limited by either smaller of size or pcb_ptr -> size */
   /* change blocked process state to WTR queue */
}
...
}
```

Note that in the above example, it is assumed that the blocked process (waiting for a message) maintains the location and size of its message area in its PCB. When control finally returns to *p_get_message()* (immediately after the call to *pkcall()*), the message is in the specified location.

3.3 Queue binding

A process can be associated with one or more queues (this is different from other systems, such as Posix, in which a queue can be shared by more than one process). In order to inform the kernel that a process wants to associate with a queue, it must bind to the queue. This is done by the process call the *bind()* command with the number of the queue to which the process wishes to bind. The process-level implementation of *bind()* is as follows:

```
unsigned bind(unsigned queue_no)
{
return pkcall(BIND_QUEUE, &queue_no);
}
```

The kernel must create a queue associated with this queue number and associate the process with it. If a queue already exists and is associated another process, the call to *bind()* must fail.

A queue does not exist until a process has bound to it. Messages sent to a queue that does not exist are to be discarded.

4 Applications

An application is a process performing a task determined by actions performed on a set of inputs to produce a set of outputs.

Input data comes directly, or indirectly, from other processes or one of the two devices: the UART (keyboard input) or SysTick (a clock pulse). Similarly, output data is sent to other processes or the UART (screen output).

Examples of applications include:

Time-server: A time-server is a process that can supply the current time to a requesting process. Alternatively, it could send a time-pulse message to processes requesting a "wake-up" message at regular intervals.

I/O-server: An I/O server can be responsible for directing keyboard input (from the UART) to a specific process, while output from processes can be directed to specific areas of the screen (via the UART). This implies that the I/O-server associates areas of the screen to different processes.

Generic applications: A set of generic applications could write unique information to the screen at regular intervals, indicating whether they are active or have terminated.

5 Marking

This assignment will be marked as follows:

Design

The design description must describe the algorithms and data structure associated with each of the four a major components.

Total points: 8.

Software

A fully commented, indented, magic-numberless, tidy piece of software that meets the requirements described above and follows the design description.

Total points: 12.

System Testing

A set of at least six system tests should be conducted.[5] Each test is to consist of three parts: a description of the test, a description of what the test is to achieve, a description of the test results based on specific input scenarios, and an indication as to whether the software met the objectives of the test. Demonstrating the software is *not* a system test.

Total points: 5.

6 Important Dates

Available: 26 September 2017

Due: 23 October 2017 (in lab/tutorial)

Demonstration (Acceptance testing): 23 October 2017 (during lab)

Late assignments will be penalized 1.5 points per day or fraction thereof to a maximum of 4.5 points. Assignments will not be accepted or marked if submitted two weeks after the due date.

Assignments must be successfully demonstrated before they will be accepted and marked.

7 Miscellaneous

This assignment may be done with teams of two.

[5] See System Testing Fundamentals (http://softwaretestingfundamentals.com/system-testing/) for a description and definition of system testing and other stages in the test cycle.