COE4DS4 Lab #7
Multi-tasking vs. Multi-processing in Embedded Systems

Objective

In this lab you will learn about task synchronization in a multi-tasking real-time operating system (OS) on a single processor. The two main synchronization mechanisms that are introduced employ mailboxes and queues. In the last experiment you will learn about multiprocessing with two NIOS-II processors, however without any OS support.

Preparation

- Read this document and get familiarized with the source code and the in-lab experiments
- You should refer to the PDF files in the "doc" sub-folder for detailed uC/OS-II documentation

Experiment 1

In this experiment mailboxes are employed to synchronize tasks that are dependent on each other. A mailbox allows a task or an interrupt service routine (ISR) to send a pointer-sized variable (also called a *message* and that is application-specific) to one or multiple tasks. A mailbox is created using `OS_EVENT *OSMboxCreate((void *)msg)`, where `msg` is a void pointer that initializes the mailbox content (if `msg` is `NULL` then the mailbox is empty). The message is posted using `OSMboxPost((OS_EVENT *) pevent, (void *)msg)` where `pevent` is the mailbox to which `msg` is passed. `OSMboxPend()` is used to check if a message has been posted. If no message has been posted when `OSMboxPend()` is called, the caller task is suspended until either a message is received or a user-specified timeout expires. On the other hand if the message has been posted before the pend occurs, then it will be retrieved by the caller task (and the mailbox will be cleared). As it is the case with the other OS events, the caller task can perform also an `OSMboxAccept()` which will not cause a suspension if the message was not posted (an error code will be returned and the caller task execution will continue depending on how this code is parsed).

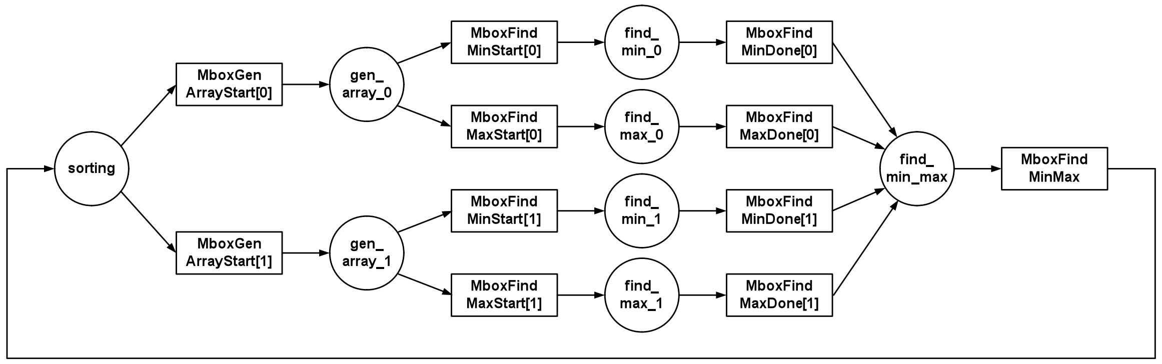

Figure 1: Task dependency graph showing inter-task communication using mailboxes.

The application in this experiment is simple and it can be explained using the task dependency graph shown above. There is a sorting task that initiates two tasks that generate two random arrays. This sorting task is also in charge of "allocating" memory through uC/OS-II-specific memory-management OS services. Note, the array generation tasks are suspended until they receive a message in their respective mailboxes from the sorting task. Once the min/max values in each array are computed (using four different tasks), the min/max values from both arrays are computed using a find_min_max task and passed back to the sorting task for printing. To monitor the runtime on-chip, performance counters are employed in the source code.

In the lab you will have to modify the reference code such that only two tasks are used to compute the min/max values (one for each array) and then two values from each task are passed to find_min_max.

Experiment 2

In this experiment, in addition to mutexes and mailboxes, flags and queues are employed to synchronize tasks that are dependent on each other. Flags are used to have a task wait for a combination of conditions (i.e., events or bits) to be set (or cleared). The application can wait for any condition to be set or cleared or for all the conditions from a flag group to be set or cleared. If the events that the calling task desires have not been posted yet, then the calling task is suspended until the targeted conditions are satisfied or a user-specified timeout expires. A message queue is used to allow tasks or ISRs to send messages to one or more tasks. A queue is created using `OS_EVENT *OSQCreate((void **)start, UBYTE size)` where `start` is the base address of the message storage area (declared as an array of void pointers) and `size` is the number of entries of the message storage area. The main purpose of message queues is to "buffer" multiple messages when a producer task is preparing data faster than a consumer is able to process the data. As it is the case other data structures of type `OS_EVENT *`, OS services such as pending, accept, deletion, … can be performed on the queues.

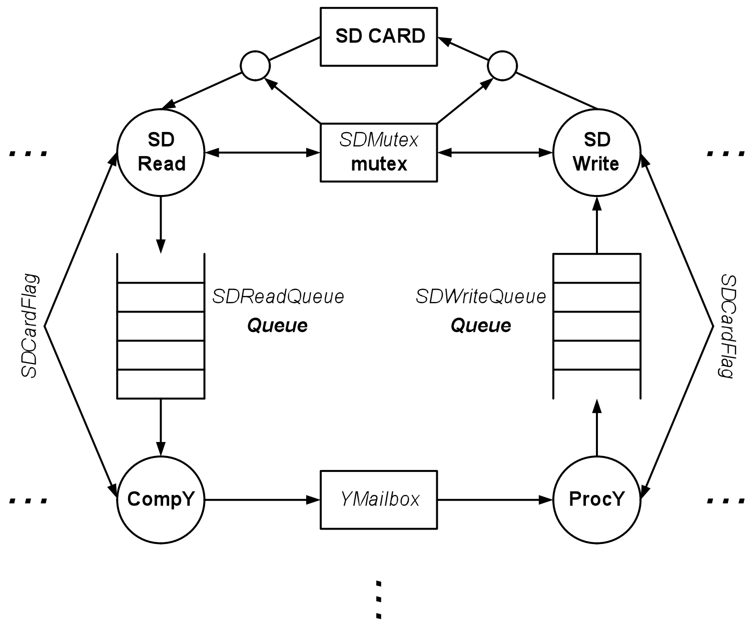

Figure 2: Task dependency graph showing inter-task communication using queues.

In this experiment queues are used to pass the data between the SD card and the processing tasks. SDCardFlag is a multibit flag used to keep the tasks synchronized. Comp(ute)Y tries to read from queue SDReadQueue and if it is empty sets a flag within SDCardFlag to get SDRead to fill the queue with data. Once full, SDRead indicates back through SDCardFlag that CompY can continue. As CompY finishes computing a set of Y values, it passes a pointer to the completed Y values to Proc(ess)Y. As with SDReadQueue, ProcY and SDWrite cooperate in filling/emptying SDWriteQueue through a flag within SDCardFlag. Both SDRead and SDWrite require the physical SD card resource, which is guarded by software mutex (called SDMute). The ellipsis in the above figure indicates that multiples of each of these blocks can be present in the system. Specifically, a number of tasks which read from/write to the SD card can be implemented, as can multiple tasks for computing/processing on both the read and write sides, which can communicate through multiple mailboxes.

In the lab the SDRead and SDWrite tasks have hard coded reading/parsing/writing of BMP images of size 640x480. CompY assigns the red component to the luma value (Y) in of `tasks.c` (line 528). Y computation should be updated in-lab with the correct equation (refer back to labs 1 and 4 for the RGB to Y conversion equations). In the given implementation, the main function of ProcY is to post data to the SDWriteQueue. However, understanding its behavior and how it is synchronized with the other tasks is essential for performing more complex functionality, as it will be the case in the take-home exercise.

Experiment 3

The uC/OS-II implementation for the Altera's NIOS processor provides OS support only for a single central processing unit (CPU). If multiple NIOS CPUs are employed, the users must manage their code.

In the figure shown below, we illustrate how two NIOS processors (CPU1 and CPU2) communicate through shared memory. In the system description defined in Qsys, CPU1 is connected to the host terminal through the joint test access group (JTAG) universal asynchronous receiver transmitter (UART) and it stores its instruction/data in the external static random access memory (SRAM); CPU2 is connected to the Character liquid crystal display (LCD) and it stores its instruction/data in the field-programmable gate array (FPGA) embedded memory. Both CPUs share a memory block (mapped also onto the FPGA embedded memory). The access to this shared memory is arbitrated through a hardware mutex block, which is defined as an component available in Qsys. CPU1 sends messages to the Character LCD and CPU2 sends messages to the terminal through the shared memory.

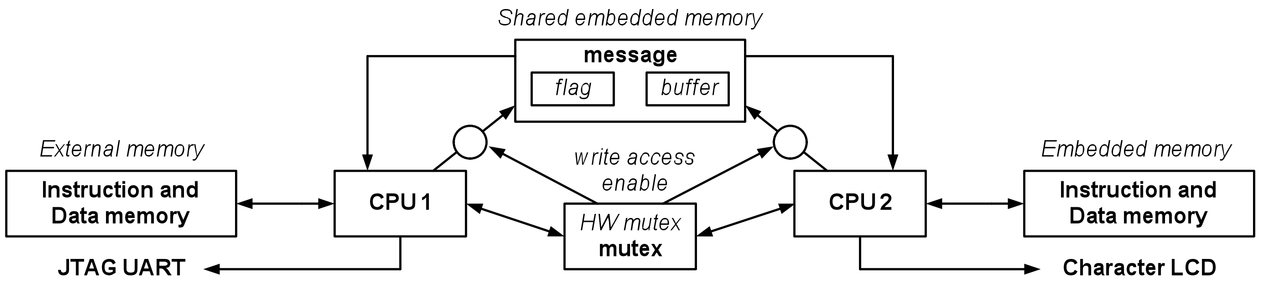

Figure 3: Using shared memory and hardware mutex for synchronizing two NIOSII processors.

Step-by-step instructions to set up the NIOS II software development kit (SDK) are provided in the *readme_toolflow.txt* file. Although a single system has been built in Qsys and the Verilog sources are compiled in Quartus to a single bitstream file, for each of the two CPUs the software source code needs to be compiled independently. This will result in two separate executable files that are loaded in their independent memory spaces (external memory for CPU1 and embedded memory for CPU2).

Because there is no OS running, keeping track of time in software is done using functions provided by Altera, such as `alt_ticks_per_second()` and `alt_ticks()`. Using an user-defined parameter for the number of milliseconds (`MS_DELAY`), we can control how often we sent a new message from one CPU to the shared memory, from where it will be read in order to be printed to the corresponding display device (JTAG UART for CPU1 or Character LCD for CPU2). For a message to be posted in the shared memory, first the hardware mutex needs to be locked; if the lock was successful, a check is done to ensure that no other messages have been posted by either CPU; until the current message is cleared, no new messages will be posted. Regardless of the number of ticks that have passed since the last post, in the infinite loop the message flag from the shared buffer is read to double-check if a new message has been received for printing. Note, this `message->flag` is read without locking the mutex because both CPUs need to access the same single memory location. Regardless which CPU wins the arbitration for the address port of the shared memory block (done in hardware at the Avalon bus layer), the flag that is read from the single location of the shared memory will be passed to both CPUs. Even though both CPUs execute concurrently the machine code from their memory space, only one of them will continue to access the shared memory block, in order to process the message (`message->buf`), depending on the content of the `message->flag`.

In the lab you are asked to modify the reference code such that depending on the configuration of the input switches, CPU1 can send messages either to Character LCD or the terminal and CPU2 can send messages either to the terminal or the Character LCD; because the system description (Qsys) the 8 least significant switches are connected to CPU2 and the next 8 switches are connected to CPU1, you can choose the switch from each group that can control where CPU1/CPU2 send their messages. Although not used in the in-lab experiment, a couple of push buttons are connected to each CPU in order to extend the functionality in the take-home exercise.

Write-up Template

COE4DS4 – Lab #7 Report
Group Number
Student names and IDs
McMaster email addresses
Date

There are 3 take-home exercises that you have to complete within one week. Label the projects as exercise1, exercise2 and exercise3.

Exercise 1 (2.5 marks) – Modify *experiment1* as follows.

Two random generation tasks will generate two distinct two-dimensional arrays of size 32x32 each (there will be a total of 1,024 elements in each of the two arrays). The elements from each of the two arrays should be represented as 32-bit *signed* integers, however the values for the first array A should be in the range -10,000 to +10,000 (inclusive), while the values for the second array B should be in the range -400 to +400 (inclusive).

Compute the two-dimensional array C =A x B in four distinct "calculation" tasks as follows. The first calculation task uses as input the top half of the rows from A and the left half of the columns from B (and hence it computes the top-left quarter of the elements from C); the second calculation task uses as input the bottom half of the rows from A and the left half of the columns from B; the third calculation task uses as input the top half of the rows from A and the right half of the columns from B; the fourth calculation task uses as input the bottom half of the rows from A and the right half of the columns from B. The elements of C should be represented also as 32-bit *signed* integers.

The results of the four calculation tasks (i.e., the top-left, bottom-left, top-right and bottom-right quarters of C) will be "merged" together and all the 1,024 values from C should be printed to the terminal. The printing should be from the top line to the bottom line and from the leftmost element to the rightmost element within each line. Please note, because the buffer used by `printf()` may overflow if 1,024 values are passed to it (without any delays to let the UART unit empty its content to the terminal), you must "slow down" the printing using either `OSTimeDly()`or `OSTimeDlyHMSM()`; it is up to you to learn how often and for how much time you need to suspend the task that prints the 1,024 values before you resume it.

You must use performance counters to measure the time from the beginning of random generation until the 1,024 values from array C are ready to be printed. You should break the total time into the time required for random generation and array calculations respectively and give a concise interpretation of the performance numbers in your report.

Submit your sources and in your report write a third-of-a-page paragraph describing your reasoning.

Exercise 2 (2.5 marks) – Extend *experiment2* as follows. You should read from the SD card two color BMP images (*image1* and *image2*) of sizes 640x480 each and you should write back to the SD card a single grayscale BMP image (*image3*) of size 640x480 as follows. The first 320 columns from each line of *image3* are the luma (Y) values of the odd pixels from the corresponding line from *image1*; the last 320 columns from each line of *image3* are the Y values of the even pixels from the corresponding line from *image2*. Note, the SD card driver that is provided facilitates multiple files to be open at the same time. You can assume that all the images have positive height. It is up to you to decide how to partition the functionality into multiple tasks; nonetheless you must follow the "pipe-flow" described in the figure from *experiment2*. Submit your sources and in your report write a third-of-a-page paragraph describing your reasoning.

Exercise 3 (4 marks) – Modify *experiment3* to implement the multiplication of two 32x32 arrays `A` and `B` (with the same representation and input ranges as requested in the problem statement for **exercise1**) as follows. CPU1, which is connected to the external SRAM, and hence it has more memory space than CPU2, will generate the two random two-dimensional arrays (1,024 values each) when either of the buttons connected to CPU1 is pressed (note, as it can be noticed from the source code, two push buttons are connected to each of the CPUs). Once the random array generation is completed on CPU1, if either of the push buttons connected to CPU2 has been pressed, then half of the calculations, i.e., multiplications and additions required to compute half of the elements from `C =A x B`, will be done on CPU1 concurrently while the remaining calculations for the other half of the elements from C will be done on CPU2. Subsequently all the 1,024 values from C should be merged on CPU1 and printed line by line to the terminal.

The choice is yours to decide how to partition the load for computing the elements of `C` on the two CPUs, so long as only half of the elements of `C` are computed on each CPU. It is apparent that “some” of the input values from arrays `A` and `B` respectively need to be passed from CPU1 to CPU2 and “some” of the result values from array C need to be passed from CPU2 to CPU1 (message passing should be done through the shared memory, as done in-lab for *experiment3*). The choice is, again, yours to decide which values need to be passed between the two CPUs. Note also, depending on your specific choice to partition the load on the two CPUs, as well as the capacities of the two embedded memories connected to CPU2 (i.e., shared embedded memory and the instruction/data embedded memory), it might be necessary to transfer smaller sets of data between the two CPUs on several occasions - the onus is on you to exercise your judgment on this matter.

After the sorted data is printed to the terminal, if either of the push buttons connected to CPU1 has been pressed again, then a new pair of random arrays is generated … and so on … the same behavior described above will be repeated. If push buttons connected to CPU1 are pressed multiple times while the random array generation/array calculations/printing is happening, their effect will be taken into account only once after all the above steps have been completed. The same principle is applied also to the push buttons connected to CPU2.

On CPU1 only, use performance counters to measure the time from the start of random array generation until the 1,024 values from array C are ready to be printed. Interpret the results and compare them against the results from **exercise1** (the individual times required for random array generation, array calculations, data transfer between CPUs and merging of the values from array C should be discussed). You must provide a concise explanation in your report for the results that you have observed.

Submit your sources and in your report write a third-of-a-page paragraph describing your reasoning.

VERY IMPORTANT NOTE:

This lab has a weight of 9% of your final grade. The report has no value without the source files, where requested. Your report must be in “pdf” format and together with the requested source files it should be included in a directory called “coe4ds4_group_xx_takehome7” (where xx is your group number). Archive this directory (in “zip” format) and upload it through Avenue to Learn before noon on the day you are scheduled for lab 8. Late submissions will be penalized.