ECE 4501/6501: Advanced Embedded Systems

Mini Project 4
RTOS with Blocking Semaphores and Priority Scheduling
Due Date: Friday, Nov. 12, 2021, 11:59 PM
(Accept Until: Nov. 16, 2021, 11:59 PM)
(Estimated Number of Hours: > 10 hours – Difficulty Level: 4)

In this mini project, you will extend your RTOS kernel to include blocking semaphores and implement fixed and dynamic priority scheduling algorithms. You will also perform experiments to assess the performance of the tasks running on your RTOS.

Go to the following link to accept the Mini Project 4 assignment in the **GitHub Classroom**: https://classroom.github.com/a/T4RNRwcC. You will get access to a private repository created for you in the @UVA-embedded-systems organization on GitHub which contains the starter files for the mini project. You should use this repository for developing and testing your code. Your submission will also be to this repository.

We have added one more periodic task and one more aperiodic task to run in your RTOS:

- **Task 6** is the second periodic task which includes: i) the **`PeriodicUpdater`** background thread that is triggered by a periodic timer and increments a counter; ii) the **`Display`** foreground thread that continuously reads the counter value and displays it on the lower part of LCD.
- **Task 7** is the second aperiodic task that is triggered by pushing S2 Button on the BSPMKII and adds the **`Restart`** foreground thread to the system. The Restart thread kills/stops other foreground threads running in the system, sleeps for 50 ms, and then restarts the foreground threads and resets their global variables.

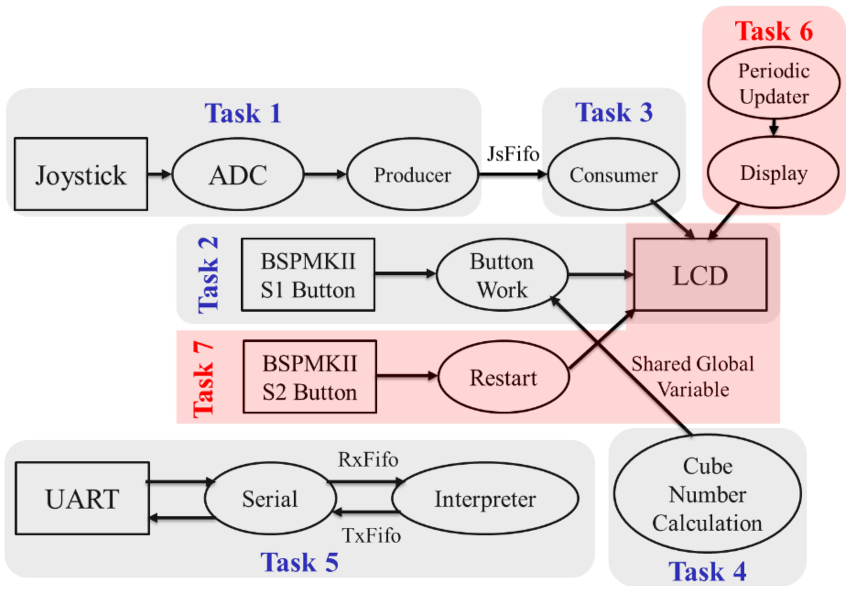


Figure 1: Data Flow Graph for Mini Project 4

Remember that functions **OS_AddSW2Task** and **OS_AddPeriodicThread** are for adding aperiodic and periodic threads to RTOS. The **priority** parameters in the **OS_AddThread, OS_AddPeriodicThread, OS_AddSW1Task,** and **OS_AddSW2Task** functions allow the user to specify the relative priority of the threads. The priority of background threads does not affect the fact that all background threads (ISRs) can preempt any foreground thread.

Mini Project Parts

Part 1 (Performance Measurement) In this part, you will modify **os.c** and **Main.c** files to measure some performance parameters.

A – In **os.c,** you are provided with a new set of variables (defined in the TCB structure) for saving the arrival time, wait time, and execution count of each thread.

```
uint32_t ArriveTime;
uint32_t WaitTime;
uint32_t ExecCount;
```

ArriveTime is the time that the thread is added to the system. **WaitTime** is the time elapsed since **ArriveTime** of a thread until it is first scheduled to execute in the system. **ExecCount** is the number of times a thread is executed and should be incremented whenever the thread is scheduled to run. Use the timing mechanisms implemented in Mini Project 2 to measure these times for a thread and save them in its TCB. Your changes should only be in **os.c**.

B – In **Main.c**, you are provided with global variables **DisplayCount** and **ConsumerCount** for measuring the performance of **Display** and **Consumer** threads. They should be incremented whenever these threads write to the LCD.

```
unsigned long DisplayCount;
unsigned long ConsumerCount;
```

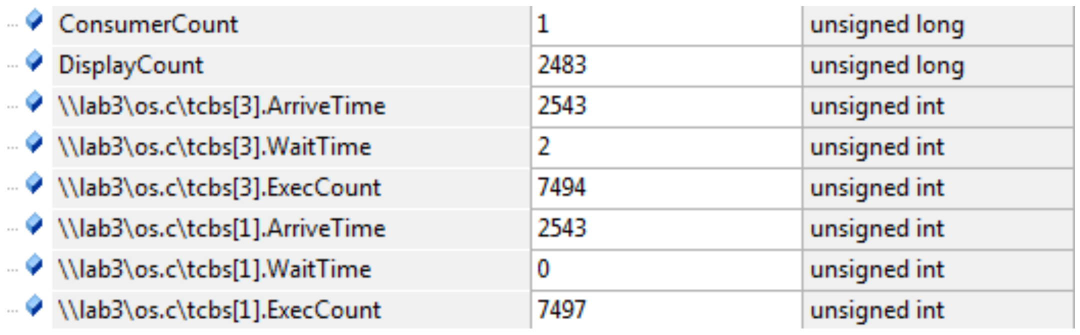
C – In **Main.c**, you are provided with global variables **Button1RespTime** and **Button2RespTime** for measuring latencies of Task 2 and Task 7. This latency is defined as the time between a push and the response on the LCD.

```
unsigned long Button1RespTime;
unsigned long Button2RespTime;
```

Note: You need to figure out where are the right locations for calling the time functions and incrementing count values within **os.c** and **Main.c**.

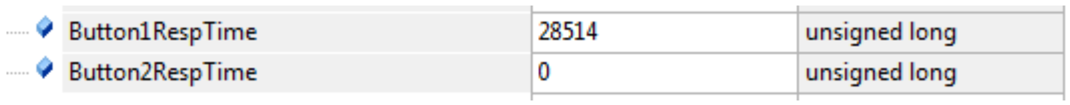
Experiment 1: Run the system without pushing any of the buttons on BSPMKII. Make a table showing the arrival time (**ArriveTime**), wait time (**WaitTime**), and execution count (final value of

ExecCount) for the **Consumer** and **Display** threads as well as their performance parameters (the final values of **ConsumerCount** and **DisplayCount**). You may use the Watch window in Keil debugger as well as the interpreter commands to observe the values of these parameters after the system has stopped running. An example is shown below:



ConsumerCount	1	unsigned long
DisplayCount	2483	unsigned long
\\lab3\os.c\tcbs[3].ArriveTime	2543	unsigned int
\\lab3\os.c\tcbs[3].WaitTime	2	unsigned int
\\lab3\os.c\tcbs[3].ExecCount	7494	unsigned int
\\lab3\os.c\tcbs[1].ArriveTime	2543	unsigned int
\\lab3\os.c\tcbs[1].WaitTime	0	unsigned int
\\lab3\os.c\tcbs[1].ExecCount	7497	unsigned int

Experiment 2: Repeat running the system this time by pushing Button 1 and Button 2 on the BSPMKII and measure the latency or response time of Task 2 and Task 7 by recording the value of **Button1RespTime** and **Button2RespTime** parameters. An example is shown below:



Button1RespTime	28514	unsigned long
Button2RespTime	0	unsigned long

Deliverables for Part 1:

- Modifications to **os.c** and **Main.c** to measure:
  - **ArriveTime**
  - **WaitTime**
  - **ExecCount**
  - **ConsumerCount** and **DisplayCount**
  - **Button1RespTime** and **Button2RespTime**
- **Snapshot** of the parameters measured for Experiment 1 in the Keil debugger's Watch Window.
- **Snapshot** of the parameters measured for Experiment 2 in the Keil debugger's Watch Window.
- Question 1: What can you say about the responsiveness of the tasks 2, 3, 6, and 7? Provide an explanation on the behavior you observe during these experiments.
- Question 2: What happens if you make the **Display** thread sleep for 1 ms? Try this and explain your observation.
- **BONUS Question 3:** What happens if we convert to a cooperative scheduling scheme in which all the foreground threads call **OS_Suspend()** after each execution? Try this, explain your observation, and relate to your answers to Question 1 and Question 2.

◆

Part 2 (Blocking Semaphores) In this part, you will add implementations for blocking semaphores. In Lecture 13, we learned about two possible implementations for blocking semaphores. In the first implementation, we simply add a `blockPt` field to the TCB. If the `blockPt` is Null, the thread is not blocked. If the `blockPt` contains a semaphore pointer, it is blocked on that semaphore. The scheduler will skip blocked threads when looping over the TCB linked list.

This simple implementation will not allow you to implement bounded waiting. However, it is okay to use this simpler implementation in this mini project. The `blockPt` is defined for you in the TCB structure in `os.c`, but you need to modify the `OS_Wait`, `OS_Signal`, `OS_bWait`, `OS_bSignal` and `Scheduler` functions according to the blocking semaphore implementations presented in Lecture 13.

Note: You may keep the previous implementation for cooperative spinlock semaphores in these functions but use the macro `blockSema` to easily switch between spinlock and blocking semaphore implementations. Note that when `#define blockSema` is commented out in the `os.c`, `blockPt` field will not be added to the TCB and any code for blocking semaphores should be replaced with the code for the spinlock semaphores.

Experiment 1: Test your blocking semaphore functions using the `Testmain6` function in `MiniProject4Test.c`. Print the values of counters (`SignalCountX` and `WaitCountX`) in the Watch window inside Keil debugger and observe the number of waits and signals. You may also observe this in the PuTTY terminal using the interpreter commands.

Experiment 2: Repeat Experiment 1 but with the cooperative spinlock semaphores enabled.

Remember: You can only have one `main` function in your project. To run the `Testmain` functions, you should only add `MiniProject4Test.c` to (and remove the given `Main.c` from) your Keil project. Also, for testing each part, you should rename the relevant `TestmainX` function to `main`.

Deliverables for Part 2:

- Your code.
- **Snapshot** of the parameters measured for Experiment 1 in the Keil debugger's Watch Window or the PuTTY terminal.
- **Snapshot** of the parameters measured for Experiment 2 in the Keil debugger's Watch Window or the PuTTY terminal.
- Question 1: Do you see any difference between the results of experiments 1 and 2? Why?

◆

Part 3 (Fixed Priority Scheduler) Implement the basic fixed priority scheduler given in Lectures 16-17 which always schedules the highest priority thread to run next. When there are two threads of the same priority in the active list, this scheduler reverts to the round robin scheme. You only need to change the implementation of the `OS_AddThread` and `Scheduler` functions. The priorities are statically assigned to all the threads and they are fixed in this mini project.

Note: You may keep the implementation for the round-robin scheduler as the default mode in your scheduler but use the macro `prioritySched` to easily switch between round-robin and priority scheduling algorithms.

Experiment 1: Test your fixed priority scheduler by running `Testmain2` function in `MiniProject4Test.c`. Print the values of counters, the start time, wait time, and the execution count of the threads in the Watch window inside Keil debugger.

Experiment 2: Repeat experiment 1 and 2 in Part 1 but this time using the blocking semaphores and the fixed priority scheduler.

Deliverables for Part 3:

- Your code.
- **Snapshot** of the parameters measured for Experiment 1 in the Keil debugger's Watch Window.
- **Snapshot** of the parameters measured for Experiment 2 in the Keil debugger's Watch Window.
- Question 1: Do you see any difference between the results of Experiment 1 and those observed when running `Testmain2` with a round-robin scheduler (in Mini Project 2)? Explain the reason for the observed behavior.
- Question 2: Do you see any difference between the results of Experiment 2 with those observed in experiment 1 and 2 in Part 1? Explain your observations.

◆

Part 4 (Dynamic Priority Scheduler) Write the implementation for a dynamic priority scheduler with aging (presented in lectures 16-17 and Valvano's textbook). Specifically, you need modify the implementation of the `Scheduler` and `Timer2AHandler` functions. You may use the three variables `age`, `FixedPriority`, and `WorkPriority` provided to you as part of the TCB structure in `os.c`.

Note: You may keep the implementation for the round-robin and fixed priority schedulers but use the macro `aging` whenever feasible to easily switch to the dynamic priority scheduler. Feel free to modify the macros or add new variables.

Hint: Your implementation needs to periodically increment the age of those threads that are not blocked or not sleeping. You can increase the priority of threads that have not run for more than 8 milliseconds.

Experiment 1: Test your new scheduler by running `Testmain2` function in `MiniProject4Test.c`. Print the values of counters and the start time, wait time, and execution count of the threads in the Watch window inside Keil debugger.

Experiment 2: Repeat the experiment 1 and 2 in Part 1, but this time using the *blocking semaphores* and the *dynamic priority scheduler with aging*.

Deliverables for Part 4:

- Your code.
- **Snapshot** of the parameters measured for Experiment 1 in the Keil debugger's Watch Window.
- **Snapshot** of the parameters measured for Experiment 2 in the Keil debugger's Watch Window.
- Question 1: Do you see any difference between the results of Experiment 1 and those observed when running **`Testmain2`** with the *fixed priority scheduler* in Part 3? Explain the reason for the observed behavior.
- Question 2: Do you see any difference between the results of Experiment 2 with those observed in experiments 1 and 2 in Part 1 (when using the *spin-lock semaphores* and the *round-robin scheduler*)? Explain your observation.
- **BONUS Question 3:** Do you see any difference between the results of Experiment 2 with those observed in Experiment 2 of Part 3 (when using the *blocking semaphores* and the *fixed priority scheduler*)? Explain your observation.

◆

Mini Project Deliverables and Grading

The table in the next page lists the deliverables and their corresponding points for this mini project. Commit and push your changes to your private repository on **GitHub** before the submission deadline. Include a PDF report containing all the deliverables (any calculations, snapshots, answers to questions, and links to videos). The first page of your report should include your name and computing ID. You may upload any required videos to a platform such YouTube or Google Drive. Make sure the videos are public or accessible to those with the link.

Note that the latest commit before the deadline will be considered your submission. Any major changes to your submitted code and report after the deadline and before posting the grades will be considered a late submission.

Survey: After completion of this mini project, please go to the following link and complete the survey: https://virginia.az1.qualtrics.com/jfe/form/SV_29V4M2CVnDEctBI. (Copy the link into browser)
This will be anonymous feedback but will be counted towards your class participation. Attach a **snapshot** of the completion page to your project report.

Deliverables	**Points**
1) Your code for the following functions:	
• `OS_Wait`	5
• `OS_Signal`	5
• `OS_bWait`	5
• `OS_bSignal`	5
• `Scheduler`	15
• `Timer2AHandler`	10
2) Deliverables for **Part 1**	
• Modifications to `os.c` and `Main.c` to measure:	
○ `ArriveTime`	5
○ `WaitTime`	5
○ `ExecCount`	5
○ `ConsumerCount` and `DisplayCount`	5
○ `Button1RespTime` and `Button2RespTime`	10
• **Snapshot** of the parameters measured for Experiment 1	5
• **Snapshot** of the parameters measured for Experiment 2	5
• Question 1	10
• Question 2	5
• **BONUS** Question 3	(+5)
3) Deliverables for **Part 2**	
• **Snapshot** of the parameters measured for Experiment 1	5
• **Snapshot** of the parameters measured for Experiment 2	5
• Question 1	5
4) Deliverables for **Part 3**	
• **Snapshot** of the parameters measured for Experiment 1	5
• **Snapshot** of the parameters measured for Experiment 2	5
• Question 1	5
• Question 2	5
5) Deliverables for **Part 4**	
• **Snapshot** of measuring parameters for Experiment 1	5
• **Snapshot** of measuring parameters for Experiment 2	5
• Question 1	5
• Question 2	10
• **BONUS** Question 3	(+10)
Total	160
(BONUS – Optional)	(+15)