SCHOOL OF PHYSICS,
ENGINEERING AND TECHNOLOGY

Systems Programming for ARM
Laboratory Scripts

Dr. Andy Pomfret

Laboratory 10

Sleeping and Waiting

10.1 Aims

In this laboratory session, you will:

- Modify the DocetOS kernel and scheduler to support sleep requests from tasks
- Add `wait` and `notify` support to the kernel
- Create a recursive mutex that can control access to shared resources

You will be expected to apply consistent coding standards (indentation, naming, comments etc.) to the code that you generate during this lab. You may also wish to keep this code, if it works well, for submission as part of your assignment.

Make a copy of your saved project from last week. You could call it `DocetOS_sleep_wait`, perhaps. Use Project→Open Project... to open this new copy.

Remove from the project the SVC delegate that you created last time, including the enum constant in `os.h` and the import and jump table entry in `os_asm.s`. Remove the `mode_utils` files and source group. Leave the implementation of `list_remove()` that you created in the scheduler, though.

Despite its new name, this project (as you know) does not yet have sleeping or waiting support. Your tasks today will be to add these.

10.2 Sleeping

There is currently no way for a task to request that it should be able to 'sleep' (avoid being run) for a certain length of time. Your first task will be to add this functionality.

Create a new source file called `sleep.c` in the `src/OS` folder, and add it to the OS source group. Create a header file `sleep.h` (in `inc/OS`) to go with it. Inside the header file, add guards as follows:

```
#ifndef SLEEP_H
#define SLEEP_H

#endif /* SLEEP_H */
```

Remember that everything you add to this header file should go between the `#define` and the `#endif` directives.

Include `OS/sleep.h` and `OS/os.h` in `sleep.c`. Include the line

```
#define OS_INTERNAL
```

before the place where `os.h` is included to ensure that you gain access to the internal API.

Include `<stdint.h>` in `sleep.h`. Create a function called `OS_sleep()`, which takes a `uint32_t` as its argument and does not return anything. Put its prototype in `sleep.h`, and its definition in `sleep.c`. This function should:

- Store the time that the task has asked to be woken into its TCB. You will need to create a field in the TCB to hold this information (in `scheduler.h`). You might choose to create a generic `data` field that could also be used for other things. Note that the function `OS_currentTCB()` returns a pointer to the running task's TCB, and that `OS_elapsedTicks()` returns the total number of pre-emption ticks that have occurred since the system was last reset.
- Set a flag in the TCB's `state` field to indicate that the task is asleep. The neatest way to do this is to use a `#define` to define `TASK_STATE_SLEEP` to refer to a bit in the field. Put this `#define` in `scheduler.h` along with the existing `TASK_STATE_YIELD`.
- Call `OS_yield()` to initiate a task switch.

Now modify the code in `scheduler.c`.

- Find the scheduler callback function. Locate the point where it checks to see if a particular task exists, and returns it if it does. This is the part you'll need to

modify.

- The scheduler should not return a task that's asleep. Add an `if` statement to check the `TASK_STATE_SLEEP` bit.
  - If the `TASK_STATE_SLEEP` bit is clear, the task is runnable and can be returned as before.
  - If the `TASK_STATE_SLEEP` bit is set, the task is asleep. Check its wake time, held in the TCB's `data` field. If it is lower than the current time, clear the `TASK_STATE_SLEEP` bit and return the task; otherwise, leave it asleep.[8]


You should also add some kind of a check to ensure that if *all* tasks in the list are asleep, the scheduler doesn't just loop until one wakes up. The idle task should be returned instead. How can you achieve this?


Replace the code in the two tasks in `main.c` so that both contain an infinite loop, within which there's a call to `printf()`. Add a call to `OS_sleep(100)` in the loop in task 1 and a call to `OS_sleep(20)` in the same place in task 2. Build and run the code. Does it behave as you expect?


Play with different sleep times. Can you estimate the time interval between system ticks? In other words, how long an interval does `OS_sleep(1)` represent?


This mechanism for implementing a timed sleep is not terribly efficient, because sleeping tasks are left in the same list as runnable tasks and the scheduler has to pass over them. Removing sleeping tasks from the round robin, and reinserting them when they wake, would be much better but is also more complex. Some kind of *priority queue* implementation to hold sleeping tasks in order of their wake times would probably be the most efficient implementation, so that the scheduler could know with a single check whether there were any tasks to wake at each context switch.

10.3 Adding `wait` and `notify` support

The second half of the lab involves creating an implementation of a reentrant mutex. But before you can implement a mutex, you must add some functionality to DocetOS to allow a task to enter a waiting state, and to be notified when the thing that it's waiting for takes place.

This is not a trivial task, and will require some careful thought! **The implementation of wait and notify in this script is not particularly efficient**, and could be improved in a few ways as we'll see later, but will provide the necessary building blocks for a good, scalable implementation.

The implementation will involve **removing** waiting tasks from the round robin, into a

[8] What will happen when the current time overflows its storage and rolls back to zero? What can you do about this?

separate list, and then restoring them to the round robin when they're notified. The tricky part of this is that the wait and notify mechanisms must interact with the round robin – and what happens if a context switch takes place during the modification?


To start off, remove the calls to `OS_sleep()` in the tasks. Verify that the two tasks fight for control of the serial port and that the messages are interleaved.

10.3.1 Thread safety


Open `scheduler.c`. Find the static `list_add()` and `list_remove()` functions that are used to add and remove tasks from the round robin, and note the comments above them: they are not currently thread-safe. In other words, if two tasks are trying simultaneously to modify the round robin, corruption of the list could occur. Similarly, if a task is trying to modify the round robin and a context switch occurs, the context switch itself relies on iterating through the round robin and could encounter list corruption.


This is because each list modification consists of multiple steps, and the list is in an inconsistent state until the steps are all complete. It is **not possible** to make these functions thread-safe in their current form using the `LDREX` and `STREX` instructions, because there is no single point at which the list modifications become active.

It's worth noting that this is mainly because the list is doubly-linked. In a singly-linked list, it is usually possible to ensure that a list modification can be committed in a single step. This is particularly true if the list is used like a stack, and additions and removals take place only at the head. Consider your memory pool implementation, which essentially maintained a singly-linked list of unallocated blocks that behaved like a stack. Adding a block to the list consisted of two steps: first new the block was made to point to the existing pool head, and then the pool head was updated to point to the new block. Only the second of these steps resulted in a change to the list; prior to this the only writes had been to the new block, which was not part of the list at the time. It would be reasonably simple to use `STREX` for this, and go back and retry both operations on failure. A similar argument applies to removal of items.

Making the task list singly-linked is not really a solution, because the double-linking makes removal of a task from an arbitrary point in the list much more efficient.

However, we can make the following observations:

- **Only tasks will ever need to use the "wait" mechanism**. It doesn't make sense for handler mode code to attempt to wait, or for a handler to try to force a running task to wait asynchronously.
  This permits the use of an SVC delegate for implementing the wait mechanism, that will call the existing `list_remove()` function. An SVC delegate is immune from being interrupted by other tasks or by the context switch due to its elevated

priority. It will also never itself interrupt the context switch, because the SVC interrupt is synchronous – it is triggered by the `svc` instruction and can be triggered only from thread mode.

- **The "notify" mechanism should be available to ISRs as well as tasks**, to support the implementation of semaphores that can be released by ISRs.
  This means that an SVC delegate is not appropriate here, because it cannot be invoked from handler mode, and it *can* be interrupted by other ISRs.
- **The context switch is a safe place to implement round robin modifications**, as long as it (and SVC delegates) are the only places these modifications are performed.
  If we can contrive a way to allow the *scheduler* to add notified tasks back to the round robin before identifying a task to return, this will be safe.

So, the waiting and notification mechanism in this lab will make use of **three** lists: the existing round robin, a singly-linked list of waiting tasks, and a singly-linked list of "pending" tasks that have been notified but not yet added back to the round robin.

The "wait" mechanism will be implemented in an SVC delegate, and will remove the current task from the round robin and add it to the waiting list. The "notify" mechanism will remove a specified task from the waiting list and add it to the *pending* list. The scheduler will then iterate through the pending list and add each task it finds back to the round robin, having first removed it from the pending list.

We will have to take care to ensure that the operations on the singly-linked list are safe against concurrent modification, particularly in the case that an ISR interrupts either the scheduler or a task that's attempting to wait, and notifies one or more tasks.

Recall that to use `LDREX` and `STREX` from `C`, you will need two CMSIS compiler-specific primitives:

```
uint32_t __LDREXW (uint32_t *addr);
uint32_t __STREXW (uint32_t value, uint32_t *addr);
```

The `__LDREXW()` primitive causes the compiler to generate an `LDREX` instruction. The parameter is the address to load from, and the primitive returns the loaded value. The `__STREXW()` primitive causes the compiler to generate an `STREX` instruction. The parameters are the value to store and the address at which to store it, and the primitive returns the `STREX` return code: zero for success and one for failure.

To use these primitives, you will probably have to do some type casting – the pointer arguments are all `uint32_t *`, and the non-pointer arguments and return types are all `uint32_t`.

10.3.2 Waiting

Open `scheduler.c`, and:

- Define another task list, just below `task_list`, called `wait_list`. Initialise it in the same way.
- Add two new functions near `list_add()` and `list_remove()` called `list_push_sl()` and `list_pop_sl()`. These will use `LDREX` and `STREX` to provide thread-safe operations. For example, `list_push_sl()` could be written as

```
static void list_push_sl(OS_tasklist_t *list, OS_TCB_t *task) {
  do {
    OS_TCB_t *head = (OS_TCB_t *) __LDREXW ((uint32_t *)&(list->head));
    task->next = head;
  } while (__STREXW ((uint32_t) task, (uint32_t *)&(list->head)));
}
```

  The `list_pop_sl()` function should take a single parameter of type `OS_tasklist_t *` and return an `OS_TCB_t *`.
- Add an SVC delegate, perhaps called `_OS_wait_delegate()`, to `scheduler.c`. Prototype it in the appropriate place in `scheduler.h`, add an enum constant to `os.h`, and add an import and jump table entry to `os_asm.s`. Add a line in `os.h` to create a macro called `OS_wait()` that invokes the `_svc_0()` helper. The SVC delegate should:
  - Call `list_remove()` to remove the current task from the round robin
  - Call `list_push_sl()` to add the current task to the wait list
  - Set the PendSV bit to invoke a context switch (see the end of `_OS_taskExit_delegate()` for example)

You should be able to test your `OS_wait()` implementation at this point. Adding a call to `OS_wait()` inside one of your tasks should cause that task to be suspended immediately, while the other(s) should continue to run. In the debugger, if you put a breakpoint inside the scheduler and inspect the `task_list` and `wait_list` variables, you should see one of your TCBs enter the wait list while the other remains in the round robin.

10.3.3 Notification

Notification must be done in a standard function, not an SVC delegate, so that it can be used from inside ISRs as well as by tasks.

Right now, because there is only one wait list but potentially multiple mutexes, the only possible action is to notify *all* waiting tasks whenever notification is required, because all waiting tasks are mixed in together. This is inefficient because tasks waiting for other reasons will be notified, and will run again only to find that their required resource is still unavailable. Later, we will look at ways to improve this.

Again, in `scheduler.c`:

- Define *another* task list, just below `wait_list`, called `pending_list`. Initialise it in

the same way.

- Add an `OS_notifyAll()` function. The prototype (in `scheduler.h`) should be

  ```
  void OS_notifyAll(void);
  ```

  and the function should repeatedly call `list_pop_sl()` to remove tasks from the wait list, and then `list_push_sl()` to add them to the pending list, until the wait list is empty.
- You will now need to add some code to the scheduler itself. When it starts running, it should repeatedly call `list_pop_sl()` to remove tasks from the pending list, and then `list_add()` to add them to the round robin, until the pending list is empty.


It is possible that the loop in the scheduler could be interrupted by an ISR that adds something to the pending list, but this should be safe: either the list modification takes place at a *different* time from the call to `list_pop_sl()`, in which case the next pop will respect the newly-added list head; or it will take place at the *same* time as the call to `list_pop_sl()`, in which case the `STREX` will fail and `list_pop_sl()` will retry.


It is very important that you understand how and why the implementation presented here is safe from concurrent modification. If you're not totally sure, please ask!


You should now be able to test your `OS_notifyAll()` implementation, at least from inside a task. Add a call to `OS_notifyAll()` inside a different task from the one that called `OS_wait()`, perhaps after a `for` loop containing a `printf()` statement. You should see that when `OS_notifyAll()` is called, the waiting task resumes. Test this in the debugger as well to make sure that you can observe the changes to the wait list, pending list and round robin.

10.4 Writing a Reentrant Mutex

You will now use your wait and notify functions to build a reentrant mutex.


First, create a new type to represent a reentrant mutex. Declare it in a new file, `mutex.h` (remember to add guards), which should be in `inc/OS`. It should have two fields: one is a pointer to a TCB, and the other a `uint32_t` counter. Call the new type `OS_mutex_t`.

Add a `#define` to create a static initialiser for the mutex. It should be a compound literal that sets both fields to zero, so that when your mutex is finished you can declare one by writing e.g.

```
OS_mutex_t mutex = OS_MUTEX_STATIC_INITIALISER;
```

Next, create `mutex.c` in `src/OS` and add it to the OS source group. In it, with corresponding prototypes in `mutex.h`, create the two functions for manipulating mutexes: `OS_mutex_acquire()` and `OS_mutex_release()`.

The `OS_mutex_acquire()` function must do the following things:

1. Use `LDREX` to load the mutex's TCB field
2. If it's zero:
   a) Use `STREX` to try to store the current TCB in the field
   b) If this fails, go back to step 1 and try again
3. If it's not zero, and not equal to the current TCB:
   a) Call `OS_wait()`
   b) Go back to step 1 and try again
4. Increment the counter field in the mutex

The `OS_mutex_release()` function must do the following things:

1. Decrement the counter field in the mutex
2. If it has reached zero:
   a) Set the TCB field to zero too
   b) Call `OS_notifyAll()`

For extra safety, at the cost of a slight performance reduction, the `OS_mutex_release()` function could also first check that the mutex is owned by the task that is calling it.

10.4.1 Testing

Implement the following test code:

- Import `mutex.h` into `main.c`
- Declare a mutex as a (static) global variable
- Initialise the mutex in `main()`
- Just before each `printf()` statement in the two tasks, insert a call to `OS_mutex_acquire()`
- Balance it with a call to `OS_mutex_release()` just after each `printf()` statement

Build and run your code. Does your mutex work correctly?

If you see only one task running, this is not necessarily a bug. Each of your tasks is currently obtaining the mutex, printing some stuff (which is time-consuming), and then releasing the mutex again before **immediately** looping back round and acquiring it again. The proportion of the time that the task spends *with* the mutex is hugely greater than the proportion of the time that it spends *without* it. So if the other task starts to run and tries to acquire the mutex, it will rarely succeed. This is kind of a pathological condition; in general tasks will not behave like this, and there won't be a problem. But if you like you can add a call to `OS_yield()` inside `OS_mutex_release()` if you want to guarantee a

context switch when the mutex is released.


Depending on your implementation, you may find either that the mutex appears to work, or that deadlock occurs after a small amount of run time. Either way, there is a bug in this implementation. We'll now look at that bug and see how it can be prevented.

10.5 Improving `wait` and `notify`


There is a significant design weakness in the current implementation. If a task finds that a mutex is not available, it will call `OS_wait()`. But if, by chance, a context switch happens *just before* this call, and the mutex is released, then the task will still call `OS_wait()` and will be waiting for the release of a mutex that has already been released. This is the cause of the deadlock you may have just seen, and it must be avoided.

One possibility is to use a *check code*. A check code is a code that is obtained prior to starting an operation, and then checked to make sure it hasn't changed just as the operation is finished. If the check code has changed, it's an indication that something could be wrong, and the operation should be re-tried. In this case, the check code could be a code that is changed every time the `OS_notifyAll()` function is called (for example a simple counter, incremented on every notification). A task that might need to wait first obtains the check code, and then determines whether or not a wait is required. If it is, the check code is passed to the `OS_wait()` function, and if the check codes don't match, a notify must have taken place in the meantime and `OS_wait()` simply returns without changing the task state. This is known as *fail-fast* behaviour: the call to `wait` will quickly fail whenever there's a possibility of deadlock, even if deadlock is not assured.


Implement a simple check code. You will need to:

- Create a new static global variable in `scheduler.c` to act as a notification counter, ensuring that it is initialised to zero
- Write a new function (with a prototype in `scheduler.h`) to return the notification counter
- Add a line to `OS_notifyAll()` to increment the notification counter, **before** iterating the wait list
- Change the definition of the `OS_wait()` pseudo-function to accept an argument of type `uint32_t` (and to use the `_svc_1()` helper)
- Change the `_OS_wait_delegate()` delegate to use the stacked value of `r0` as the passed check code: if it does not not match the current value of the notification counter, abort the wait and return without modifying the task status

Now use this in your mutex implementation: get the notification count before checking if the mutex is available, and if it isn't, pass the count to `OS_wait()`. Remember to do this in such a way that when `OS_wait()` returns (either immediately due to the check failing,

or due to a notification) the notification count is obtained again before the whole thing is retried.


Build and test your code. Does it fix the problem, if you observed a problem previously?


You will need to observe the behaviour of your code **very carefully** in the debugger to be sure that it works properly. For example, if you make a mistake and the check code *never* matches the notification counter, then `OS_wait()` will never wait. But there is no visible effect to this; it just means that instead of waiting, tasks will enter a "spinlock", repeatedly checking to see if the resource they need is available. This is horrible for efficiency of CPU usage, but at a glance looks like it's working. Once you've checked yourself and you think it's working, it's probably best to ask when you get to this point and I can come and check with you!

10.6 Improving Efficiency


Neither the sleeping implementation, nor the mutex implementation, is particularly efficient as it stands. But some reasonably small tweaks could help a lot:

- Sleeping tasks could be removed from the round robin, and reinserted when they wake. This could be done quite easily now you have the `list_push_sl()` and `list_pop_sl()` functions, though you'd have to take care to ensure that `OS_sleep()` could not be interrupted by a context switch while it was removing a task from the round robin. Perhaps reimplementing it using an SVC delegate would be sensible.
- If sleeping tasks are kept separate, they could be stored sorted by wake time. Either an insertion-sorted list or a binary heap could do this, with tradeoffs in code size and complexity, scalability, and maximum speed. Then the scheduler would only have to check the first task in the list to know whether anything needed waking at each tick.
- Waiting and notification is currently inefficient because the only way to notify *one* task is to notify *all* tasks. This situation could be improved enormously by maintaining lists of waiting tasks inside mutexes (and semaphores, and anything else that uses the wait/notify mechanism). By having one list per mutex, when a mutex becomes available it would be necessary only to notify *one* task from the head of the mutex's list.

These ideas are presented for information only. It is not intended that you work on any of them now!

10.7 Summary and Conclusions

By now, you should:

- Have a version of DocetOS with working sleep functionality, as well as kernel support for `wait` and `notify`
- Have a working reentrant mutex, and the knowledge you need to write binary or counting semaphores
- Understand some of the race conditions that must be considered when designing systems-level code

If you have any questions about what you have learned, please ask.

Remember to copy your project files to a network drive before you log off. This is particularly important again this week because you may wish to use this project as a basis for your assessed work.