ME241 – Introduction to Computer Structures & Real-Time Systems

Lab Project 2: The Scaffold of our RTOS

Prepared by:

Nathan Tang (20890522)

Eddy Zhou (20873917)

Submitted on:

21 September 2022

Part 1: Thread Implementation

A visual representation of GizmOS's thread implementation is shown below.

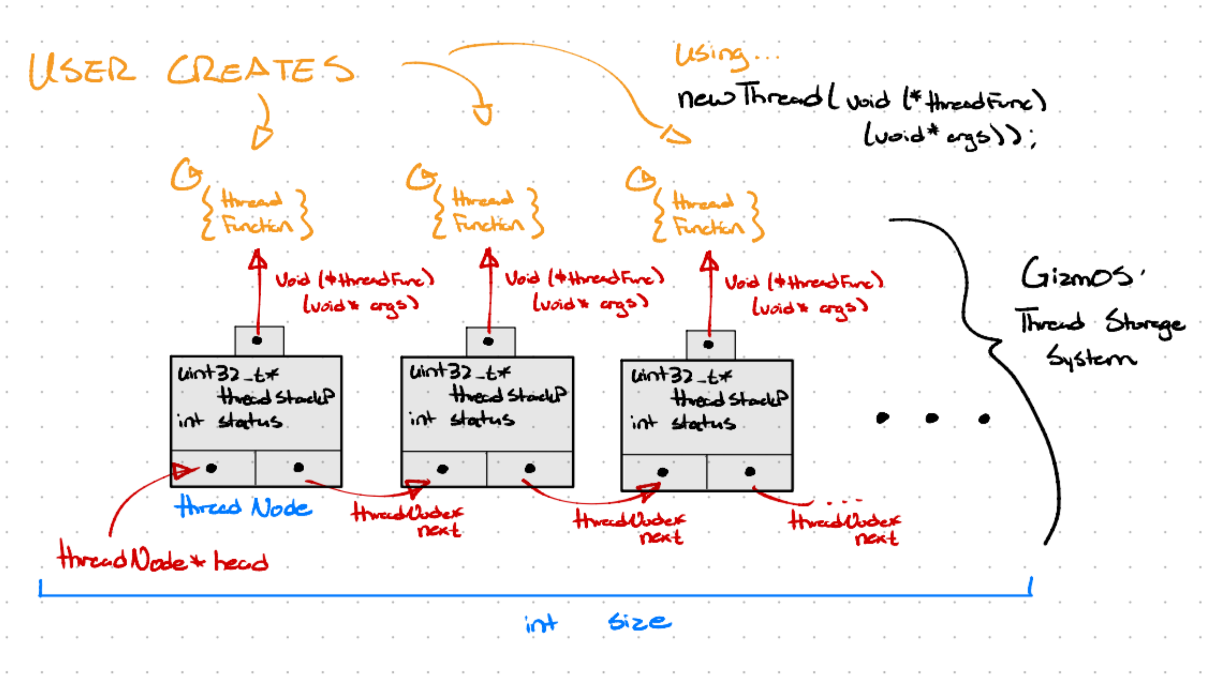

USER CREATES
using...
newThread(void (*threadFunc)(void* args));
{ thread Function }
{ thread Function }
{ thread Function }
void (*threadFunc)(void* args)
void (*threadFunc)(void* args)
void (*threadFunc)(void* args)
uint32_t* threadStackP
int status
thread Node
threadNode* next
threadNode* head
GizmOS' Thread Storage System
int size

Figure 1 High-Level Diagram of GizmOS Thread Storage System

In `osDefs.h`, the maximum number of threads are defined to be 5 (`MAXNUMTHREADS`) along with the stack size for both MSP (`MSR_SIZE`) and PSP (`PSP_SIZE`). Each thread created will also be in one of the 4 defined states (`CREATED, WAITING, ACTIVE, and DESTROYED`).

The threads will be dynamically allocated and stored in a singly linked list data structure. At the expense of precise memory management, the decision to dynamically allocate the threads was to reduce the amount of memory the OS uses at a given time. To represent a single node, a struct was used with members listed in the following *Table 1*. To represent the linked list itself, a struct was used with members listed in the following *Table 2*.

`struct threadNode`	
`uint32_t* threadStackP`	Stores the thread stack pointer of the thread.
`void (*threadFunc)(void* args)`	Stores the pointer of the function to be run when the thread runs.
`int status`	Stores the status of the thread (such as if it is created, destroyed, etc).
`struct threadNode* next`	Stores the pointer to the next node in the list.

Table 1. Thread struct for Linked List Node and its characteristics

struct threadLinkedList	
`threadNode* head`	Stores the pointer to the head of the linked list.
`int size`	Stores the size of the linked list.

When `kernelInit(void)` is called in `_kernelCore.c`, it allocates memory for an instance of the linked list using `malloc(sizeof(threadLinkedList))` and assigns it to the external pointer variable `list` for later use when adding and manipulating threads. This makes sense since the memory storing the threads should be allocated before the kernel is initialised. The head of the list and size are defined appropriately (`NULL` and `0` respectively) to avoid undefined behaviour.

A new thread can be created in `p1_main.c` using `newThread(void (*threadFunc)(void*args))` defined in `_threadsCore.h`. The following function accepts a pointer to the function that will be executed when the thread is running. In `newThread`, memory is allocated for the new thread using `malloc` and the thread function is assigned to the thread node. In addition, a pointer to the new thread stack is assigned using `getNewThreadStack` and the appropriate offset based on the defined MSR size and the number of threads currently stored.

If any thread nodes exist at the time of the current thread creation, the new thread node is added to the head of the linked list and the rest of the thread nodes are shifted further down. Otherwise, the thread node just becomes the head of the linked list. Threads can continually be added into the linked list until the number of threads reaches the maximum `MAXNUMTHREADS`. The cap on the number of threads avoids the situation when the heap runs out of memory which would result in undefined behaviour. Once all the threads are added, the kernel is started, placing the Cortex M in thread mode. This means that the thread stored in the head of our `threadLinkedList` is pushed onto the stack, as well as registers xPSR, PC, LR, R12, and R3-R0 in that particular order. Subsequent threads are then processed in some specific sequence via context switching.

Part 2: Context Switching

As a quick idea of context switching, a context switcher is used to determine what thread to process next when the previous thread decides a switch is necessary. A context switch may also occur from interrupts, but we will only focus on context switching between threads that the user makes. A high-level understanding of the Context Switcher's round-robin configuration is shown below.

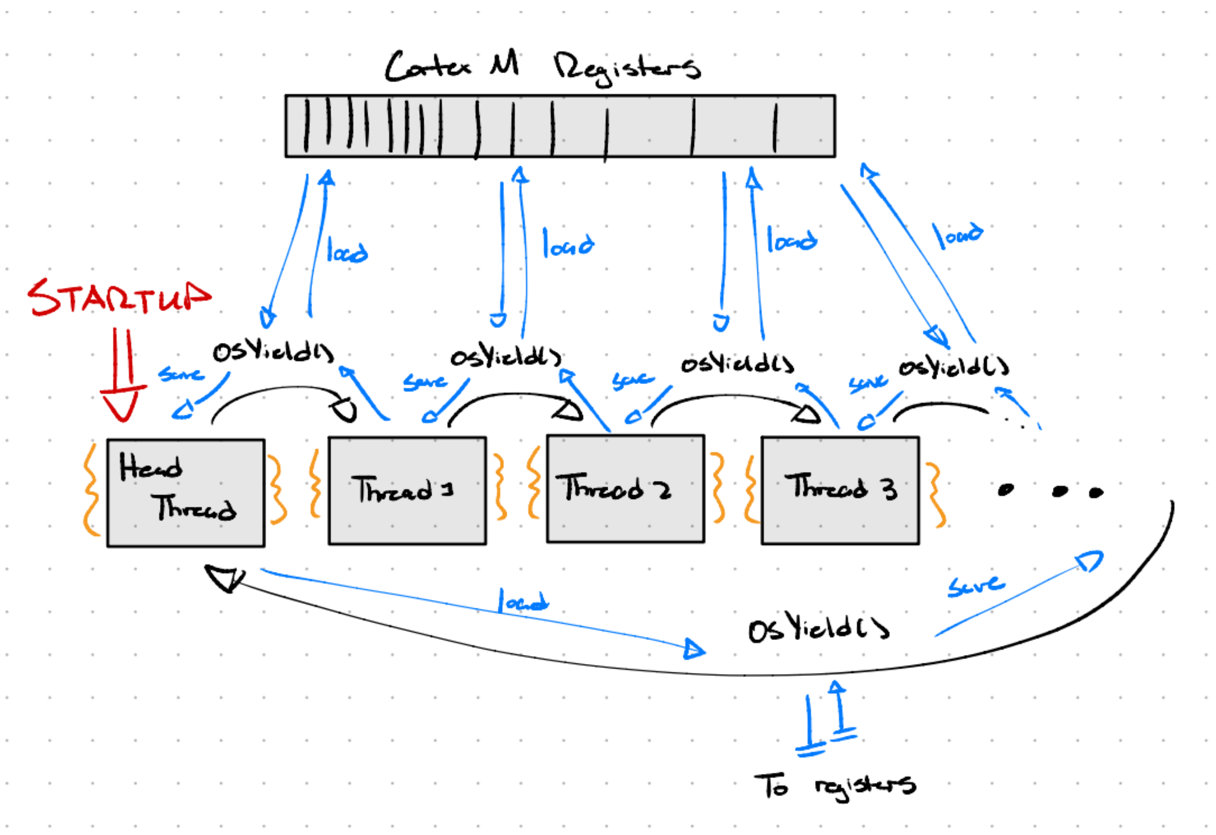


Figure 2 High-Level Diagram of GizmOS Round-Robin Scheduler

Following the creation of new threads in `p1_main.c`, the kernel is started through `osKernelStart()`. This function, defined in `_kernelCore.h`, first checks if any threads are created in the first place. Should no threads exist, `osKernelStart()` will simply return, and the main code terminates in an endless loop. Otherwise, to avoid bootstrapping, `osKernelStart()` will initially set the thread at the head of `threadLinkedList` to run. This is done by setting the PSP register to the PSP of the thread[1] via `__set_PSP((uint32_t)list->head->threadStackP)`. Following setting the PSP register, `osKernelStart()` will call `osYield()` for the first time.

[1] Doing so stops a fault from occurring in the first `MRS r0,PSP` of svc_call.s because the PSP is initially empty. By setting the PSP with the PSP of the first thread, we ensure that the PendSV interrupt will always have a value of PSP to load into r0.

By setting `tasknum` to -1 in `osKernelStart()`, the first ever call of `osYield()` will simply increment `tasknum` up by 1 and pend a context switch to the head thread by setting the `PENDSVSET` control bit to 1 in the Interrupt Control and State Register (ICSR) followed by a flush of the pipeline with `__asm("isb")`. This context switch triggers our PendSV interrupt in `svc_call.s` which contains the assembly code needed to switch to the `tasknum`th thread (on initial startup this is the 0th thread, or the head of the `threadLinkedList`). This assembly code first stores r4-r11 of the current thread, grabs the new PSP, loads the new thread's past r4-r11, and branches back to thread mode. Once back, we are now running in the head thread where we can now begin context switching between all the other threads.

Subsequent calls of `osYield()` are done inside the thread functions themselves. When `osYield()` is called, GizmOS switches between threads in a round-robin manner. That is, the next thread to be pushed onto the stack is the thread next in the linked list, wrapping back around to the head thread when the tail of the linked list is reached. This is done by incrementing `tasknum` up by 1 on every call of `osYield(),` setting `tasknum` back to 0 when it reaches the size of the `threadLinkedList`

.

Aside from switching threads according to some `tasknum`, `osYield()` also stores the current thread's PSP inside its node in the thread linked list. A thread's PSP location will change as we switch between other threads. As a result, the thread's current PSP location should be remembered so that we don't revert back to the wrong PSP location (a lot could go wrong otherwise). Because `__get_PSP()` stores the value of the next thread's stack[2], we have to store `__get_PSP() - 16*4` in the `threadStackP` of the current thread's node.

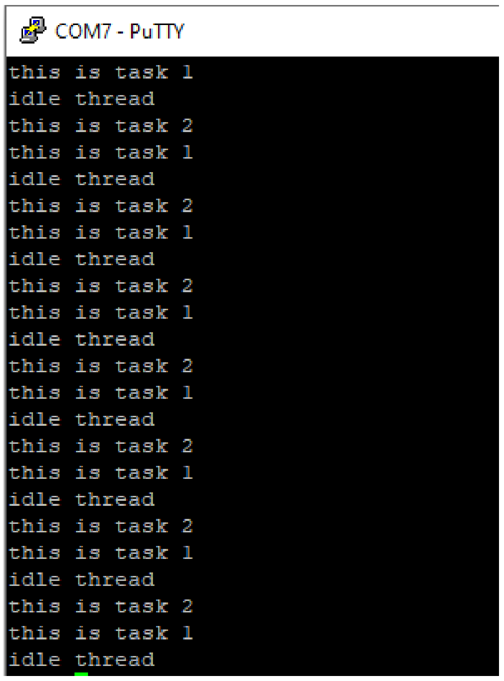
To update the status of each of the threads, their status is set in `osYield()` where the current thread is changed from ACTIVE to WAITING and the next thread is changed from WAITING to ACTIVE.

Bringing this together, the `osYield()` function first traverses to the current thread's node (indexed by `tasknum`). It then stores the current thread's PSP location inside the node, sets the current thread's status to WAITING, and increments `tasknum` along with updating the current thread node accordingly`.` By this point, `tasknum` and `currentNode` are referring to the next thread, and so we set the next thread's status to ACTIVE with `currentNode->status = ACTIVE`. After this, `osYield()` triggers the context switch by setting the `PENDSVSET` control bit to 1 in the ICSR.

[2] This is because of the way GizmOS' task counter is set up. The PSP was stored before triggering the PendSV Interrupt.

Part 3: Testing

To test the RTOS, three threads were created in `main` in `p1_main.c`. Each of the three threads runs a simple void function that prints a single line before switching to the next thread. The following *Figure 1* shows the output in the PuTTY console.



```
this is task 1
idle thread
this is task 2
this is task 1
idle thread
this is task 2
this is task 1
idle thread
this is task 2
this is task 1
idle thread
this is task 2
this is task 1
idle thread
this is task 2
this is task 1
idle thread
this is task 2
this is task 1
idle thread
this is task 2
this is task 1
idle thread
```

Figure 1. PuTTY Console Output of RTOS

References

[1] “Tim Lab Project 2.” University of Waterloo. Accessed 20 Oct. 2022.