Time-Critical Computing Project
Phase 1 Report

Raphael BOUILHOL
Nanyang Technological University
Polytech Sorbonne
rbouilhol@hotmail.fr

Phylicia NGUYEN
Nanyang Technological University
Universite de Technologie de Troyes
phylicia.nguyen@utt.fr

I. Introduction

A task, in OS-III, is comprised around several elements, namely a function associated to the task, a stack, and a structure overseeing it all, the TCB (*Task Control Block*), also containing critical information such as the priority and the options.

The task is first created using the *OStaskCreate* function. This function is used to initialize the stack and all the field of the task TCB, especially the pointers needed for the task to exist and run.
The task is then added to the "Ready List", which is the data structure used by the scheduler (with the *OSched* function) to determine which task can be executed.
The Ready List is implemented through a Bitmap table in which we put the n-th bit at 1 when a n-th priority task is present in the Ready List. The Ready List will then present itself as an array of MAXPRIORITY size, with each index being a double linked list containing every TCB of the tasks ready at a priority of Priority = Index.

When the Scheduler is called, it simply fetches the highest priority number, then the first task available at this priority level in the Ready List.
Once it's the task's turn to get the CPU ressources, the task executes until the end of its code, at which point the task is usually deleted using the *OSTaskDel* function. This is one type of task code, the other one being the ones in which the code is mainly comprised of an infinite *while* loop, in which a OS-III function will be called to make the task wait for an event (a delay, a signal, a message...).

A wide variety of tools to create delays and to signal tasks exists, such as Semaphores and delays based on time. When using the latter, the task is put on the "Tick Wheel".
The Tick Wheel is a data structure implementing a hash table, allowing the OS to only update the time remaining before a task is release onto the Ready List for the task that are on the index *Tick Match % WHEELSIZE*, with the Tick Match being the number that the Tick Counter have to match so that the task can be released.

In this context, our project is to develop an API allowing users to make a task be released periodically onto the Ready List. In addition to this, we will also be replacing the data structures implemented for the Tick Wheel and the Ready List by a Skip List and an AVL Tree, which are supposed to be more efficient when researching an element.

II. Periodicity

A. Implementation

Firstly, we are going to describe how the periodicity is implemented.

The task is first created using the *OSTaskPeriodicCreate* function, then made periodic using the *addPeriodicTask* function. This function

first assigns the Priority of the task at the Period value, which is a choice made to respect the Rate Monotonic scheduling algorithm, in which the task with the shorter period has the highest priority. Then, the task is added to both the Skip List to start the periodic release, and the Ready List in order to have a simultaneous release of every task once the system is started.

From this point on, the task will first be executed when it will have the CPU resources, then be released again in the Ready List once the period value is reached. When this happens, the *OS-PeriodicTaskRdy* function is called. This function is used to make a periodic task ready. To do so, we simply remove the task from the Ready List, then insert the task both in the Ready List, and in the Skip List to achieve periodic release. To reduce overhead for the release, we clear reset the stack of the task 1 tick before it is released, because it can be initialize from previous executions of the task and not reseting it prevent the periodicity from working.

Finally, when the periodic task reaches the end of its code, the *endTask* must be called to remove the task from the Ready List and call the scheduler to find the next highest priority task to run.
Similarly, the API is also comprised of a *removePeriodicTask* function used to remove the periodicity aspect of a task.
Furthermore, this very aspect is also centered around the substitution of the pre-existing data structure to manage time delays, the Tick Wheel, by another data structure. We chose the *Skip List* because it is more efficient in the search for an element, used when we want to add or delete an element.

B. Skip List as Tick Wheel

The Skip List is a data structure containing several levels with sorted elements. When we add an element, it is added on the base level, then it has a probability p (here, p = 50%) of also being inserted a level higher. If it is indeed inserted a level higher, it has again p chances of being also inserted one level higher. That way, the higher you go in the levels, the fewer elements are present. That way, searching an element becomes faster by starting at the beginning of the highest level, and going down a level if we found an element higher than the one we are looking for.

In our case, the Skip List will be comprised of the TCBs of the periodic tasks, sorted by the Tick Remain value, the number of ticks remaining before the task is released. To achieve the implementation, we added a field in task TCB, NextTCB, containing the addresses of the next TCB on every possible level of the Skip List. That way, every level is a Linked List, but we only need to keep one instance of TCB for all the levels. Moreover, the number of levels can be managed dynamically, but we chose to define it statically with a value of 4.

Finally, we had to modify the way the data structure was updated when a Tick occur. Because the Skip List is sorted based on the Tick Remain value, we have to update this very value for every task every time a Tick occur so that the insertion can work, and not only update a small portion of the tasks like it was with the Tick Wheel.

With the same idea in mind, we also replaced the data structure implemented in the Ready List.

C. AVL Tree as Ready List

In this part, we will firstly be interested in how to construct an AVL tree, and then replacing the Ready list with this new structure.

Firstly, as we said previously, the Ready list is a list of all the tasks ready-to-run. It presents itself as a double linked list in which each element contains the pointers of the previous and next tasks TCB. Sorted by priority, it allows the scheduler to determine which task must be executed.

In our case, we want to replace this data structure

by a more efficient structure. This desired structure is an AVL tree. Thus, to understand why an AVL tree is efficient, we will see the main differences between a normal tree like a binary search tree (BST) and this structure.

With a BST, the height between branches of a tree can be very different, depending on the element inserted in the tree and its position in this same sorted tree. That is why if we search the last element of the longest branch, we can take a significant time compared to other branches.

Unlike the BST, an AVL tree is balanced tree. Indeed, if we subtract two any branches of the AVL tree, the height difference cannot be superior to 1. It is this feature which allows the AVL tree to be more efficient when we search for an element.

So, we must re-implement the basic functions of a tree such as insertion and deletion functions which include a searching function. To achieve this feature, we created functions which perform rotations according to all the possible cases: - Left left case - Left right case - Right right case - Right left case These functions are all detailed and explained in the source code.

After creating the AVL tree, we must replace the functions linked to the Ready list by the AVL tree functions. The most part of the code is included in the OS core, which is the main file for core functions such as Ready list.

As a first step, we decide to not directly delete the ready list but add our structure in parallel. So, we have added the AVL tree functions at the same place than Ready list functions. In other words, we have added the AVL tree functions each time the Ready list functions are called.

Then, for allowing C/OS-III to use the structure of AVL tree instead of the double linked list, we must point the AVL tree structure. That is why, we affect the TCB of highest priority task in the variable *OSTCBCurPtr*.

After these changes, we have added some other pieces of code like global variables for keeping our AVL tree or declare some types, fill the headers file with our functions and others.

However, the main changes are linked to the ready list and the replacement of this one.

III. Conclusion

To conclude, we can simply measure the performances of our implementation with the one originally present.

The original overhead time between the tick ISR and the releasing of a task has been measured at 195 by the CPU Timer, and our implementation, without the replacement of the Ready List, has been measured at 113, we successfully implemented more efficient data structures to reduce the releasing overhead.