The Design of *HAL9000* Operating System

Ardelean Octavian Ioan

Capitolul 1

General Presentation

1.1 Working Team

Specify the working team members' names and their responsibility.

1. Marian Mihai
   (a) Threads: dealt with Timer.
   (b) Userprog: dealt with Argument passing + validation of system call arguments (pointers).
   (c) VM: dealt with CCC
2. Egri Julia
   (a) Threads: dealt with Priority Scheduler.
   (b) Userprog: dealt with Process Management + file system access.
   (c) VM: dealt with CCC
3. Ardelean Octavian
   (a) Threads: dealt with Priority Donation.
   (b) Userprog: dealt with Thread management.
   (c) VM: dealt with CCC

Capitolul 2

Design of Module *Userprog*

2.1 Assignment Requirements

2.1.1 Initial Functionality

Currently, HAL9000 supports loading user-applications without passing any arguments so the first step in our project would be to add support for passing arguments to user applications. Also, multiple system calls are defined but not implemented, so for a full functionality of our project we will need to implement them.

2.1.2 Requirements

The major requirements of the "Userprog" assignment, inspired from the original Pintos documentation, are the following:

- *System Calls for Thread Management.* You have to implement the system calls *SyscallThreadExit()*, *SyscallThreadCreate()*, *SyscallThreadGetTid()*, *SyscallThreadWaitForTermination()* and *SyscallThreadCloseHandle()*.

2.1.3 Basic Use Cases

In order to understand process and thread management we can study the structure on an application, for example Visual Studio.

A complex application such as Visual Studio is composed from numerous processes, which in their turn are composed with numerous threads. We can see in figure 2.1 some of the processes that run at a given time when building a project in VS2019 and observe the different tasks they might perform: handling package servers, services or executing scripts. We can consider a process as a program in execution.

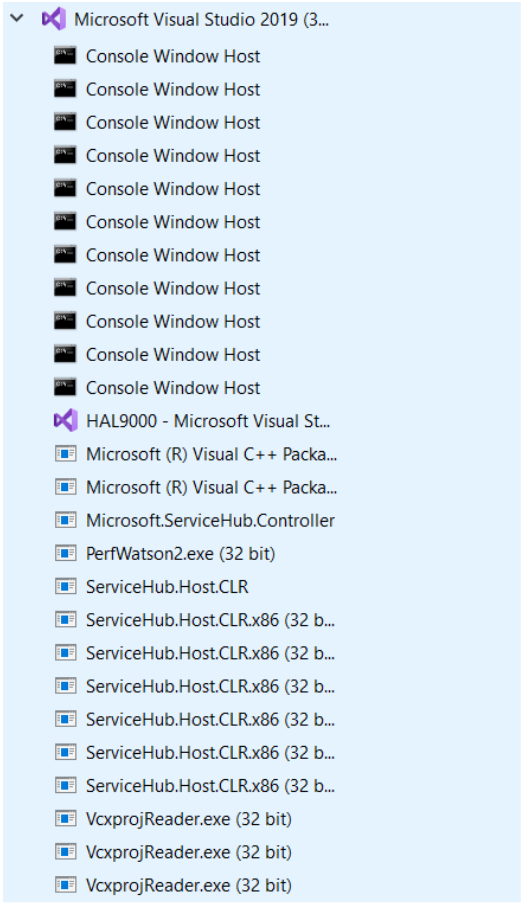

Figura 2.1: Processes of Visual Studio

Since a thread is the unit of execution within a process, we can deduce

that a process can have one or a multitude of threads. In our case, the process VcxprojReader has 4 threads (seen in figure 2.2), each with different priorities statuses and functions. These multiple threads compose a process as a whole, so if the implementation of the system calls for thread managing is faulty we will have a faulty process.

As a conclusion, a good managing of the system calls for threads is needed for the functionality of the process and, in consequence, for the functionality of the application.

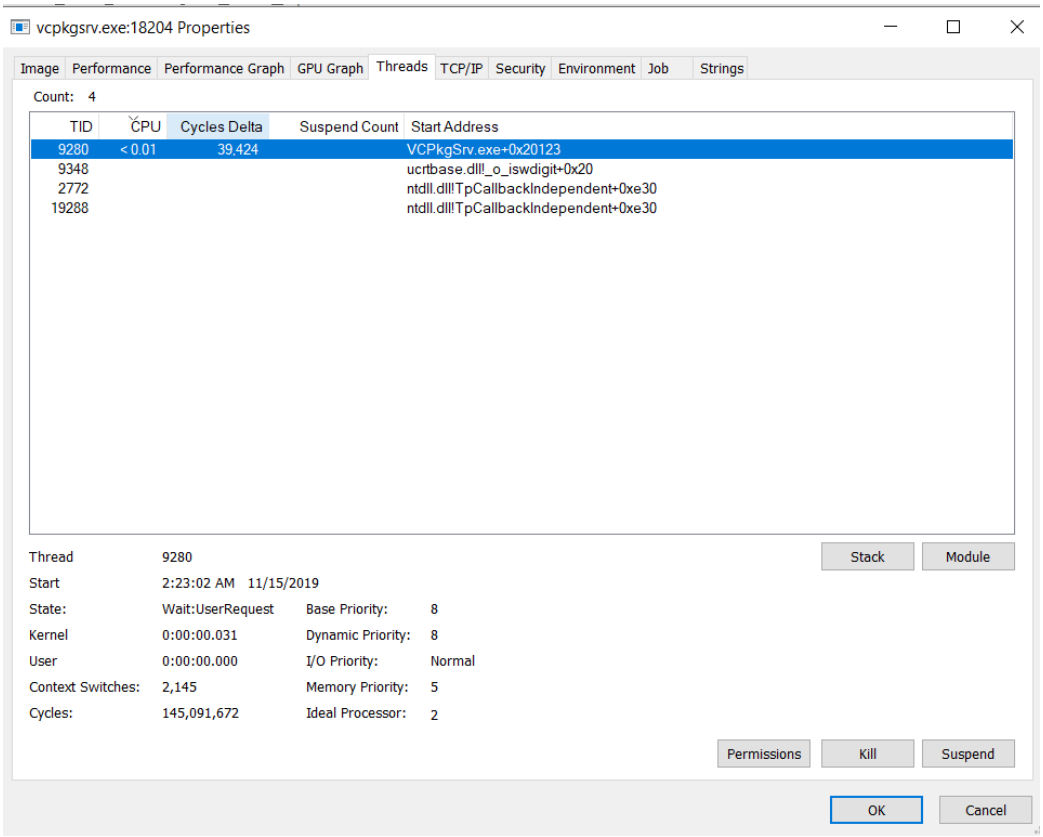

Figura 2.2: Threads of VcxprojReader process

2.2 Design Description

2.2.1 Needed Data Structures and Functions

We will need to keep track of the parent process of a thread, so in order to create a thread we will use ThreadCreateEx() instead of the function ThreadCreate() that we used in the previous assignment. This function has an additional parameter, the process to which the thread belongs.For our design, an additional paramater named ThreadHandle was needed, to keep track of the handle assigned to the thread. We will also need to implement parameter passing by modify SetupMainThreadUserStack() to properly setup the main thread's stack.

Considering how processes are formed from one or multiple threads we must keep a list of threads for each process. For this we will make use of the following functions: ProcessInsertThreadInList, ProcessNotifyThreadTermination, ProcessRemoveThreadFromList. Me must also terminate all threads of a process when the ProcessDestroy function is performed.

Lastly, we will use all the functions we were tasked to implement: SyscallThreadExit() , SyscallThreadCreate() , SyscallThreadGetTid() , SyscallThreadWaitForTermination() and SyscallThreadCloseHandle() .

2.2.2 Detailed Functionality

Compared to processes, threads are much more lightweight: faster creation, deletion, switching. Also there is much faster communication among threads which, in consequence, allow shared data structures to be maintained. However when multiple threads share the same memory locations, some problems might appear.

Seeing how threads are a part of the process, we will design our system calls for thread management with the design for system call for processes in mind. Also, for this project, we only consider issues which arise from single-thread user applications, as stated in the project specification. Also, checking if the transmitted parameters of a userprog does not fail the system

is mandatory.

Also we need to understand what a thread, a process and a handle is: a thread is part of the process, running within its own execution space and there can be multiple threads in one process. with the help of it os can do multiple tasks in parallel(depends upon the number of processors of the machine.

A handle is a generic OS term that can be a ticket to an operating system object. Each handle is unique and identifies each object. A thread is an OS object and each one you create, you get back a handle for it.

Since we have multiple functions to implement, a possible approach would be to analyze each function in particular.

1. SyscallThreadExit
   Function: Causes the executing thread to exit with ExitStatus. A possible complication to this call is when a thread has other child-threads. In this case we must terminate the parent thread may wait for the child thread to terminate.

2. SyscallThreadCreate
   When this system call is performed we will create a new thread by calling ThreadCreateEx().

3. SysCallGetId
   This function will return the id of the thread handle, or if that is invalid will return the TID. Because in our project we only use single-threaded user applications, our thread id, TID will always be equal to our process id PID.

4. SyscallThreadWaitForTermination
   Waits for process ThreadHandle to terminate and returns the termination status in TerminationStatus. We will call the ThreadWaitForTermination() function.

5. SysCallThreadCloseHandle
   We must keep in mind that losing a thread handle does not terminate

the associated thread or remove the thread object. So, when calling this function we must first check if the thread state is destroyed, then free the handle for said thread.

2.2.3 Explanation of Your Design Decisions

It's easier to keep track of all the needed changes by taking each function one by one and adding needed functionalities as the development goes on.

2.3 Tests

1. threadclosetwice - Opens a file and tries to to close it twice. The second close must fail silently or return -1.
2. threadcreatebadpointer - Passes a bad pointer to the create system call, which must cause the process to be terminated with exit code -1.
3. threadcreatemultiple
4. threadcreateonce
5. threadcreatewitharguments
6. threadexit - Checks SyscallThreadExit function
7. threadgettid - Checks the SysCallGetId function
8. threadwaitbadhandle
9. threadwaitclosedhandle - Checks SysCallThreadCloseHandle function.
10. threadwaitnormal
11. threadwaitterminated