The Design of *HAL9000* Operating System

January 30, 2021

Abstract

This document will describe and track our progress in the vast domain of operating system design. We have received a functional operating system, HAL9000, 64-bit, written for x86 symmetric multiprocessor systems. It provides some basic functionalities such as: a round robin thread scheduler, support for launching user-mode applications, a basic FAT32 file-system driver, etc. HAL9000 can theoretically run on any physical Intel x86 PC which supports 64-bit operating mode. Our focus will be in enhancing 3 main components: the thread scheduling module, the user program modules and the virtual memory mechanism.
As this is an iterative project, we will progressively add our progress here after the implementation phase of each module.

Chapter 1

General Presentation

1.1 Working Team

Our team is composed from 3 members:

1. Cristian Blaga
2. Radu Beche
3. Vlad Buda

Chapter 2

Design of Module *Threads*

2.1 Assignment Requirement

2.1.1 Initial Functionality

In the initial implementation of the Threads module, we are given an implementation of a timer. In order to use a timer, it has to be initialised first. In the initialisation step, we need to provide two important parameters, the type of the timer and the relative (or absolute) time at which it should trigger. Based on the type of the timer, we can achieve different behaviors as follows: one time triggered timer (with relative or absolute trigger time) or periodically triggered timer (with relative trigger time). After the timer has been initialised, it needs to be started. By starting a timer, any thread that has access to it can use it in order to wait for a specific trigger time. This waiting is currently implemented using a busy-waiting technique. More explicitly, this means that the waiting function, i.e. `ExTimerWait`, implements the waiting mechanism using a loop that continuously yields the thread until the time has elapsed. However, the problem with this approach is that a thread that is yielded actually goes back to a `READY` state, instead of a `BLOCKED` state. In other words, the thread will keep getting rescheduled and will keep using the CPU even if we are sure that, until the time has elapsed, it should not run.

Moreover, in the initial functionality provided, even though threads have priorities associated to them, they are not taken into consideration. This is a problem from multiple perspectives, as follows:

- The threads in the ready queue of the scheduler are all considered *equal* even if they have different priorities
- When there are multiple threads waiting for an event to signal and the event can only wake up one thread, it does not wake up the most prioritary thread
- When there are multiple threads waiting for a resource (lock for example) in a dedicated waiting queue, the first thread to be given access to that resource is not the most prioritary one

By adding a priority system for the threads, certain problems can arise such as priority inversion. Mainly, the threads with medium priority may get CPU time before high priority threads which are waiting for resources owned by low priority threads. Not having priorities at all, the current implementation does not deal with such problems.

2.1.2 Requirements

The requirements of the “Threads” assignment are the following:

1. *Timer*. Regarding this assignment, we are required to change the current implementation, i.e. behavior, of the timer in order to not use a busy-waiting loop. As it was explained in the previous section, this mechanism is inefficient when it comes to implementing a timer because it keeps the CPU busy by allowing threads (that we are sure that are blocked) to run on it. Our purpose is to maximize the proper usage of the CPU by minimizing the number of useless context switches. In order to achieve this goal, we are going to make use of the executive event mechanism (`EX_EVENT`) by blocking threads that have to wait after an event (in our case, the timer) and placing them in a dedicated

waiting queue, queue from which they will be woken up when the event signals (when the time has elapsed).

2. *Priority Scheduler — Fixed-Priority Scheduler.* We are required to modify the behavior of working with threads that have priority in order to accommodate the fact that the priority needs to be taken into consideration whenever there is to be chosen a thread from any list. It also needs to incorporate a preemptive behavior, which means that at any moment the processor will be allocated to the ready thread with the highest priority. The priority scheduler should not change in any way the threads' priorities.

3. *Priority Donation.* Given the before mentioned priority scheduling, we are required to deal with the problems that may arise, namely priority inversion. In order to solve this problem, we have to implement a priority donation mechanism. In particular, the low priority thread receives the priority of the highest priority thread waiting on the resources it owns.

4. *Advanced Scheduler (MLFQ) — Dynamic-Priority Scheduler.* This is basically a priority-based scheduler, but with variable priorities, changed by the OS (scheduler) during threads execution based on their consumed processor time and waiting time. The rules to change the threads' priorities are described in the *HAL9000* documentation. NOTE: This is required only for 4 people teams.

2.1.3 Basic Use Cases

Basic use cases for which each of the newly added functionalities may be used:

1. *Timer.*

   (a) Relative timers that have a trigger time of 0 ms should return instantly.

(b) Absolute timers with the trigger time in the past should return instantly.

(c) One (or multiple) threads waiting for a relative timer with a specified trigger time, or an absolute timer with the trigger time in the future, should all be woken up when the timer triggers.

(d) Multiple threads waiting for multiple timers with different trigger times should be woken up in an ascending order based on the trigger time they are waiting for.

2. *Priority Scheduler — Fixed-Priority Scheduler.*

   (a) Whenever a thread is to be scheduled, the most prioritary thread from the ready list must be chosen.

   (b) Whenever a thread is unblocked, if it has a higher priority than a running thread on any of the cores, it should take its place instantly (the place of the lowest prioritary one). It should not be put in the ready list,

   (c) Whenever we need to choose from a waiting list of threads, only one thread which should be woken up, we need to choose the most prioritary one.

   (d) At any moment in time, the running threads must have a priority greater than any thread present in the ready queue.

   (e) If more threads share the same priority, the threads will be chosen in a Round-Robin manner.

3. *Priority Donation.*

   (a) A low priority thread holds a resource needed by a higher priority thread to run, a switch of priorities has to occur in order to allow the system to run normally

   (b) A received priority should not be modified, but only switched in other context of other higher priority donation

   (c) A low priority thread holds multiple lock

(d) When the threads wait for each other in a chained like structure the priority donation should also take a chain form (multiple priority in versions)

4. *Advanced Scheduler (MLFQ) — Dynamic-Priority Scheduler.* TO BE COMPLETED ————

2.2 Design Description

2.2.1 Needed Data Structures and Functions

The first data structure that we are going to modify is the EX_TIMER data structure which specifies the components of a timer. The following fields will be added:

```
typedef struct _EX_TIMER
{
    ....
    volatile BOOLEAN IsTriggerTimeZero;
    EX_EVENT* ExecutiveEvent;
    ....
} EX_TIMER, *PEX_TIMER;
```

The fields added to this data structure have the following purpose:

1. *IsTriggerTimeZero.* This flag is used in order to signal that the trigger or reload time for a relative timer is of 0ms. In this case, the threads that want to wait after this timer will not be blocked at all.

2. *ExecutiveEvent.* This field is used in order to eliminate the busy-waiting technique by using its waiting list in order to queue and block the threads that are waiting for a timer to trigger (the trigger of the timer will be converted into an event signal).

The next data structure that we are going to modify is the THREAD_SYSTEM_DATA data structure which holds general information for the scheduler regarding the threads. The following fields will be added:

```
typedef struct _THREAD_SYSTEM_DATA
{
    ....
    LOCK                    RunningTimerListLock;

    _Guarded_by_(RunningTimerListLock)
    LIST_ENTRY              RunningTimerList;

        LOCK                    WaitingThreadsListLock;

    _Guarded_by_(WaitingThreadsListLock)
    LIST_ENTRY              WaitingThreadsList;
    ....
} THREAD_SYSTEM_DATA, *PTHREAD_SYSTEM_DATA;
```

The fields added to this data structure have the following purpose:

1. *RunningTimerList.* This list is used by the scheduler in order to keep track of the running timers in order to signal events when their corresponding time has elapsed.

2. *RunningTimerListLock.* Lock used in order to impose a synchronous access to the `RunningTimerList`.

3. *WaitingThreadsList.* This list is used by the scheduler in order to keep track of the blocked threads in the system and not reschedule them until they have been unblocked.

4. *WaitingThreadsListLock.* Lock used in order to impose a synchronous access to the `WaitingThreadsList`.

Next, we are going to modify the `THREAD` data structure:

```
typedef struct _THREAD
{
    ....
    LIST_ENTRY                  WaitingList;
    LIST_ENTRY                  LockWaitingList;
    ....
} THREAD, *PTHREAD;
```

The following fields have added:

- The `WaitingList` field added in this data structure serves as a pointer to the element in the list from the `THREAD_SYSTEM_DATA` data structure. It serves as a pointer which will be referenced from all the waiting lists in the program.

- The `LockWaitingList` field added serves as a pointer to the element in the list that holds all the thread waiting for a resource that the current thread holds.

The following functions have been modified or added:

1. *ex_timer.c*

   (a) *ExTimerInit.* This function has been modified in order to accommodate the new changes in the data structure of the timer. In other words, the executive event is initialized and the flag is computed.

   (b) *ExTimerStart.* This function has been modified in order to also add the timer to the `RunningTimerList` in the `THREAD_SYSTEM_DATA` data structure.

   (c) ExTimerStop. This function has been modified in order to also remove the timer to the `RunningTimerList` in the `THREAD_SYSTEM_DATA` data structure. Moreover, the event signal will be generated, all the threads waiting for it will be woken up and, nevertheless, the signal will be cleared in order to allow a correct functionality of a restart of a timer.

   (d) *ExTimerWait.* This function has been modified in order to remove the busy-waiting loop. The *IsTriggerTimeZero* flag will be used in order to perform an early check of a useless context switch. Moreover, threads that need to be blocked will be placed in both the waiting queues of the executive event and scheduler.

2. *thread.c*

   (a) *ThreadBlock*. This function will be modified in order to also add the newly blocked thread to the waiting list of the scheduler.

   (b) *ThreadUnblock*. This function will be modified in order to also remove the newly unblocked thread to the waiting list of the scheduler.

   (c) *ThreadSystemPreinit*. This function will be modified in order to also initialize the newly added fields in the `THREAD_SYSTEM_DATA` data structure.

   (d) GetThreadSystemData:

   ```
   void
   GetThreadSystemData(
       OUT THREAD_SYSTEM_DATA* ThreadSystemData
   );
   ```

   This function is created in order to allow access to the `THREAD_SYSTEM_DATA` data structure.

   (e) *ThreadUnblock*. When a thread is unblocked, find out the priority of all the threads running on all the cores. If there are threads that are running with a lower priority, choose the most unprioritary one and replace it immediately with the newly unblocked thread (the initially running thread will be placed in the ready list). Otherwise, place it in the ready list.

   (f) GetThreadPriority. This function will be modifies such that it returns the maximum values between the current thread priority and the priority of the threads in the LockWaitingList(the priority of the other threads will be acquired using this exact function; this behaviour solves the nested priority donation problem).

3. *ex_system.c*

   (a) *ExSystemTimerTick*. This function will be modified in order to periodically check the current time of the system and signal all the running timers that have their time elapsed. Moreover, if the timer is periodic then the event needs to be cleared, i.e. `ExEventClearSignal` is called.

4. *ex_event.c*

    (a) *ExEventWaitForSignal.* The pointer which will be added to the waiting list of the event will not be the ready list pointer, but rather the waiting list pointer.

5. *list.c*

    (a) *RemoveHeadList.* The current implementation of the function removes the input element from the list (the so-called "head), which may not be, in fact, the most prioritary one. Considering that we need to have access to the most prioritary element, the function will actually remove the most prioritary element.

6. *mutex.c*

    (a) *MutexAcquire.* The function will be modified such that besides blocking the current thread if the lock is acquired it will also add the newly locked thread to the waiting list of the holder thread.

    (b) *MutexRelease.* The function will be modified such that besides releasing the current lock it will also update the holder's waiting list (remove the threads that have been waiting for the lock it has released). When the holder of the lock is changed, we also need to update the new holder's lock waiting list.

2.2.2 Detailed Functionality

For the timer solution, we are going to exemplify the detailed functionality of our solution from the perspective of the life cycle of a timer. Thus, we have the following key steps:

1. *Timer initialization.* When a timer is initialized, a new timer data structure is created and its fields are set.

- Firstly, an executive event is created which has as a type the `ExEventTypeNotification` which will wake up all the threads from its waiting list when its signaled. This event type is chosen because, when a timer triggers, all the threads waiting for it will be woken up.
- Afterwards, the timer's type is set. This type describes the behavior of the timer, which can have an absolute or relative time (for the relative time, it can be one time or periodical triggered).
- Next, the timer's trigger times are set (based on its behavior).
- Finally, the `IsTriggerTimeZero` flag is set based on the time provided to the init function.

2. *Timer start.* If the timer has been initialized, the following things happen:
    - The timer is added to the `RunningTimerList` in the `THREAD_SYSTEM_DATA` data structure. The access to this list is performed synchronously using the `RunningTimerListLock` in order to avoid race conditions regarding the `RunningTimerList` because there can be multiple starting/stopping timers.
    - The `TimerStarted` field is set to true.
3. *Timer stop.* If the timer has been initialized, the following happens:
    - Firstly, the timer will be removed from the `RunningTimerList` in the `THREAD_SYSTEM_DATA` data structure.
    - Afterwards, the timer's event will be signaled in order to wake up and unblock all the threads that are waiting for the timer to finish.
    - Next, the event's signal will be cleared in order to be able to reuse the timer (to be started again).
    - Lastly, the `TimerStarted` will be set to false in order to indicate that the timer is not started.

4. *Timer uninit.*

    - Firstly, the timer will be stopped (which, implicitly, will do all the steps described previously).
    - Afterwards, it will set the `TimerUninited` field to false in order to indicate that the timer needs to be initialized.

5. *Timer wait.* If the timer has been initialized, the following things happen:

    - Firstly, if the `IsTriggerTimeZero` flag is set, then the function will return instantly, in order to avoid useless context switches.
    - If the `IsTriggerTimeZero` flag is not set, then the current running thread will call the `ExEventWaitForSignal` function which will perform the following:
        - Firstly, the CPU's interrupts are disabled in order to avoid race conditions when a timer interrupt occurs during a call to this function.
        - Next, the currently running thread will be added to the waiting list of the executive event (which has a synchronous access) in order to know which threads should be woken up when the event signals.
        - Afterwards, the `ThreadBlock` function is called which performs the following:
            * Firstly, the currently running thread will be added to the `WaitingThreadsList` from the `THREAD_SYSTEM_DATA` data structure (which has a synchoronous access).
            * Afterwards, the currently running thread will be removed from the `ReadyThreadsList` (Which has a synchronous access).
            * Finally, the scheduler is called in order to schedule another thread on the current CPU.

6. *Timer running.* If the timer has been started, as we have previously said, it will be added to the `RunningTimerList` in the `THREAD_SYSTEM_DATA` data structure. Since we have removed the busy-waiting check from the timer's waiting function, we will perform the check in the `ExSystemTimerTick` function. Because this function is called periodically (based on the `SCHEDULER_TIMER_INTERRUPT_TIME_US` constant), it makes for a perfect candidate to make the check if any timer has finished. This is because the timer's finish check is not an equality check, but rather a comparison check. Thus, in this function we will do the following:

    - We will iterate over the `WaitingThreadsList` from the `THREAD_SYSTEM_DATA` in a synchronous manner (we will acquire the lock before iterating over it).
    - If any of the timers in the list have finished, then the timers will be stopped (in case they are one time only). As we have previously said, when a timer triggers, its attached event will be signaled (the `ExEventSignal` function) and, afterwards, if the timer is periodic, its signal will be cleared. The even signal performs the following:
        - Iterates over the waiting list attached to the event.
        - For each entry in the list, it removes it and calls the `ThreadUnblock` function which performs the following:
            - It removes the given thread from the `WaitingThreadsList` from the `THREAD_SYSTEM_DATA`.
            - It inserts the given thread in the `ReadyThreadsList` from the `THREAD_SYSTEM_DATA`.

For the priority assignment, we are going to exemplify our solution based on the three different scenarios the priorities are needed:

1. *Thread schedule.*

    - Whenever a thread yields or its time slice has expired, the scheduler needs to choose another thread for the execution. The chosen thread should be the most prioritary thread in the ready list of the scheduler. However, if there are multiple threads with maximum priority, the scheduler should choose amongst them in a Round-Robin manner.

- If the currently running thread yields but it has the highest priority, it will be rescheduled.
- This behavior (choosing the thread with the highest priority and traversing the threads in a Round-Robin manner when they have the same priority) is achieved by altering the implementation of the `RemoveHeadList` function, which will find the most prioritary thread by looking for the maximum, starting from the head and iterating over the list to the tail. If the check is strictly greater, then we ensure the desired behavior.

2. *Thread access to a resource.*

   - Whenever a thread waits for a resource (a signal from an event, a lock etc.), it is placed in a waiting queue.
   - When the signal has been generated or the lock has been released, the one notified or the one that gets the access should be the thread with the highest priority (in the same manner as for the scheduler).
   - As explained previously, this behavior is achieved by altering the function `RemoveHeadList`.

3. *Thread unblock.*

   - The current implementation of the unblock method sets the thread from blocked to ready and places it in the ready list of the scheduler. If the thread has a high enough priority to run, it will be scheduled at the next timer interrupt.
   - However, if we are sure that a thread should run (that is, there are currently running threads that have a lower priority than it), then the newly unblocked thread should take the place of that

respective running thread instantaneously, without being initially moved to the ready list.

- This behavior can be achieved as follows:
  - We are going to make use of the `Interprocessor communication`, more concretely, the `SmpSendGenericIpiEx` function.
  - Firstly, we are going to define a function that will be run on all the cores of the CPU. What this function does is that it takes the currently running thread from that respective core and returns its priority as status.
  - We are going to apply the previously defined function on all the cores of the CPU and we will keep track of the minimum priority (a special case is the Idle thread; no matter the priority, the newly unblocked thread should be placed there).
  - If the newly unblocked thread has a higher priority that the minimum priority we have previously discovered, we are going to make use of the `Interprocessor communication` again in order to run this thread on that core. The initially running thread will be placed in a ready state.
  - If the newly unblocked thread does not have a higher priority than the minimum priority we have discovered, then it will be placed in the ready list of the scheduler.

For the priority donation mechanism, we are going to exemplify it based on the scenarios where it is used.

1. *Update thread's priority based on the priority donation mechanism*

   - Based on the priority donation mechanism, when a higher priority threads requires a resource currently hold by a lower priority thread, it donates its priority to that thread in order to increase the chances of releasing the resource faster.
   - This is achieved in two steps:

  – First of all, each thread was augmented with a data structure that hold all the threads waiting for the current thread to release a resource (they do not necessarily need to wait for the same resource).
  – Secondly, whenever we need to work with priority, i.e. whenever we call `RemoveHeadList`, the function `GetThreadPriority` will be called. This function is augmented in order to compute the current thread priority based on the maximum priority between the thread's priority and the priorities of the threads in its `LockWaitingList` (these priorities are accessed also using this function; this mechanism solves the nested priority donation problem).

2. *Acquire a mutex*

    - Initially, when a thread wants to acquire a lock, if that lock is already taken by another thread, it will be placed in the waiting list of that thread and its priority will not be taken into consideration.
    - Because of this, the function has been changed such that it also adds the newly blocked thread to the lock waiting list of the holder thread (in order to achieve the behaviour explained above).

3. `Release a mutex`

    - Initially, when a thread releases a lock because of the priority mechanism, that lock will be handed to the highest priority thread in lock's waiting list. However, there may still be threads waiting for that lock.
    - When this happens, the MutexRelease function will also transfer all the threads from the lock's waiting list to the waiting list of the newly holder thread. Moreover, the lock waiting list of the previous holder will also be updated such that it no longer contains the threads that were waiting for this lock.

2.2.3 Explanation of Your Design Decisions

1. *Timer design decisions.*

    - In order to remove the busy-waiting technique from the waiting function of the timer, we have decided to completely remove the checking loop from this function, add the timer to a running timer list globally available and, finally, perform the check (and, implicitly, the thread unblocking) from the timer interrupt handler associated with the scheduler (called periodically based on the `SCHEDULER_TIMER_INTERRUPT_TIME_US` constant). We have decided to make use of the interrupt time handler in order to be able to check as often as possible when a timer has finished. Moreover, since the check is not based on equality but rather on a comparison, it makes it even a better choice.
    - Since we perform both the check and the unblocking of threads in the `ExSystemTimerTick` function, which is called periodically on each core of the processor, we might have some overhead regarding the time this function takes. However, we have minimized the time spent in this function as follows:
        - Firstly, we perform the check only on the running timers. There is no need to keep track of a timer in any other state.
        - Moreover, we perform the check only on timers that have the trigger time different from 0. Timers that have the trigger time 0 are a special case which we have treated as explained in the detailed functionality. The key idea is that we have avoide useless context switches.
        - Even if all the cores of the CPU perform the check, because the access to the list is synchronous, that means that only one core (at a given moment) will stop timers and unblock threads. However, we have inserted an overhead making the global list synchronous which we can not avoid.
    - We have solved the possible race conditions that may occur as

follows:

  – If there are multiple threads that simultaneously call the wait function of the timer, they will not enter any race conditions, which might happen when trying to access the waiting lists of both the event attached to the timer and the global waiting list, because the access to the lists is synchronous (based on locks).
  – The race condition regarding the interrupt performed by the global timer during the execution of the waiting function of a timer has been taken care of by disabling the CPU interrupts (in the event wait for signal function), which means that, when a thread has started the waiting process, it can not be interrupted.

2. `Priority mechanism (including priority donation) decisions`

   - Currently, the priority mechanism is not implemented using a priority queue, but rather with a simple doubly-linked list which has as insertion time O(1) and returns the element with the highest priority in O(n). Even if the priority queue would have been much faster than this implementation, the problem with that approach is that it will not be applicable during priority donation mechanism (where there are already threads present in the priority queue and their priority updates dynamically; the queue needs to be re-balanced).
   - All the waiting lists of the mutexes, executive events, and the scheduler's ready list are updated such as they return the thread with the highest priority when the `RemoveHeadList` is called. This is ensured by the fact that the implementation of the `RemoveHeadList` has also been updated to achieve this behaviour.
   - The nested donation problem has been handled by adding the lock waiting list in the thread data structure together with an update implementation of the `GetThreadPriority` function which,

if traced, can be seen that traverses the entire dependency graph created by the priority donation mechanism when looking for the highest priority of the thread.

- We have not relied on the `ThreadSetPriority` function because our solution handles the priority mechanism solely on the modified `GetThreadPriority`.
- We have decided to keep in the `LockWaitingList` of a thread all the threads regardless of their priority (even if priority donation works with more prioritary threads) in order to allow an easy implementation of the dynamic priority donation.

2.3 Tests

We will briefly describe each test that will grade the correctness of our implementation.

1. Timer tests

   - Test if an absolute timer set 50ms in the future works properly.
   - Test if an absolute timer set 50ms in the past works properly (should return instantly).
   - Test if a relative periodic timer with a 50ms period works properly if waited on for 10 times.
   - Test if a relative periodic timer with a 50ms period works properly if waited on for once.
   - Test if a periodic timer with period 0 works properly.
   - Test if multiple threads wait for the same relative once timer with a period of 50ms.
   - Test if multiple threads (default 16) wait for a different relative periodic timer with a different period. Thread[i] waits for a timer of period 50ms + i*10ms 10 times.

- Tests to see if a relative once timer works properly for a 50ms timeout period.
- Tests to see if a relative once timer works properly for a 0ms timeout period (should return instantly).

2. Priority scheduler tests

   - Tests if multiple threads with increasing priorities waiting for an `EX_EVENT` are woken up accordingly.
   - Same test but with mutexes instead of `EX_EVENT`.
   - Tests if a highest priority thread is not de-scheduled even if it tries to yield the CPU.
   - Tests is multiple threads with highest priority are scheduled in a Round-Robin fashion.

3. Priority donation tests

   - Tests if a lower priority thread holding a resource inherits the priority of a higher one waiting for that resource
   - Same as above, but the initial thread tries to lower its priority on its own (should remain with the inherited priority)
   - Tests when the two mutexes held by a thread are requested by two other threads with higher priority. After the thread releases the first mutex, its priority should be equal with the priority of the thread currently waiting for the other lock.
   - Same as above, but the mutexes are released in different order: the second one acquired is also the second one released and the first one acquired is the first one released.
   - Same as above, but with 3 threads waiting for two mutexes. The two higher priority ones are waiting for the second mutex, while the first spawned thread is waiting on the first mutex.
   - Same as above, but the mutexes are released in different order.

- Tests nested thread scenario with mixed priority donations (with 3 additional threads).
- Same as above but with 7 additional threads.

Chapter 3

Design of Module *Userprog*

3.1 Assignment Requirements

3.1.1 Initial Functionality

For the *Program Argument Passing*, the initial functionality present in the **HAL9000** project is that no program arguments are passed to the user process. Moreover, the stack is not initialized, which means that no user program can be run.

For the *System Calls for Process Management* and *System Calls for File System Access*, none of the defined system calls are implemented. Besides implementing them, a generic handler also needs to be implemented in order to be able to uniquely identify the processes, threads and files the system calls have worked with. Moreover, all the parameters that the system calls receive need to be verified. Nevertheless, no matter the reason a process fails because of an exception, it needs to be terminated.

3.1.2 Requirements

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

- *System Calls for Process Management.* The following system calls need to be implemented: *SyscallProcessExit()*, *SyscallProcessCreate()*, *SyscallProcessGetPid()*, *SyscallProcessWaitForTermination()* and *Syscall-ProcessCloseHandle()*.

- *System Calls for Thread Management.* The following system calls need to be implemented: *SyscallThreadExit()*, *SyscallThreadCreate()*, *SyscallThreadGetTid()*, *SyscallThreadWaitForTermination()* and *SyscallThread-CloseHandle()*.
- *Program Argument Passing.* The process creation behavior needs to be changed in order to also add on the corresponding user-space stack the program's runtime arguments (given to the process on the command line, when the program is started).
- *System Calls for File System Access.* The following system calls need to be implemented in order to open existing files or create new files (*SyscallFileCreate()*), read data from a file *SyscallFileRead()* and close a file *SyscallFileClose()*).

3.1.3 Basic Use Cases

For the *Program Argument Passing*, the identified use cases are the following:

- Run a program that has **zero** arguments (except for the program's name).
- Run a program that has **exactly one** argument.
- Run a program that has **multiple** arguments.

For the **System calls** part, the identified use cases are the following:

- Create new process/thread
- Terminate process/thread
- Wait for a process/thread to finish its execution
- Get process/thread pid/tid

- Close the UM handle of a certain process/thread

For the **File handling** part, the identified use cases are the following:

- create a new file
- read from a file
- close a given file

3.2 Design Description

3.2.1 Needed Data Structures and Functions

3.2.1.1 Program Argument Passing

For the *Program Argument Passing*, no new data structures or functions are needed. However, one existing function needs to be changed, namely:

- *_ThreadSetupMainThreadUserStack*. This is the only function that needs to be changed in order to implement the program argument passing. It creates the stack in the user space where the parameters of the process are defined and can be accessed.

3.2.1.2 System calls

For both the *Process management system calls* and File management system calls, the following data structure need to be defined:

- *_UM_HANDLER_TYPE*. An enum which specifies the type of the *_UM_HANDLER* data structure described below.

```
typedef enum _UM_HANDLER_TYPE
{
    _UM_HANDLER_THREAD,
        _UM_HANDLER_PROCESS,
    _UM_HANDLER_FILE
} UM_HANDLER_TYPE;
```

- *_UM_HANDLER*. A data structure which contains the necessary fields in order to be able to manage all the processes, threads and files created and managed by system calls.

```
 // struct for holding information regarding the system call hand
typedef struct _UM_HANDLER
{
    // the ID of the resource
    UM_HANDLE              Id;

    // list entry for
    // global UM_HANDLER_LIST list
    LIST_ENTRY             UmHandlerEntry;

    //the created process
    // (in case the UmHandlerType is _UM_HANDLER_PROCESS)
    PPROCESS               PProcess;

    // the created thread
    // (in case the UmHandlerType is _UM_HANDLER_THREAD)
    PTHREAD                PThread;

    // the created/opened file
    // (in case the UmHandlerType is _UM_HANDLER_FILE)
    PFILE_OBJECT           FileHandler;

    // the process that was responsible
    // for the creation of the current UM_HANDLER
    PPROCESS               ParentPProcess;

    // current handler type
    UM_HANDLER_TYPE            UmHandlerType;

} UM_HANDLER, * PUM_HANDLER;
```

- *_SYSCALL_DATA*. A datastructure that holds a list of *UM_HANDLERs*, global across all processes, together with a lock (in order to ensure a synchronous access to the list).

```
typedef struct _SYSCALL_DATA
{
        LOCK                    UmHandlerLock;

        _Guarded_by_(UmHandlerLock)
        LIST_ENTRY              UmHandlerList;
} SYSCALL_DATA, *PSYSCALL_DATA;
```

- *m_syscallData.* A static variable that holds both the list and lock presented previously.

  ```
  static SYSCALL_DATA m_syscallData;
  ```

- *UM_HANDLE_INCREMENT.* A value that represents the increment for the *UM_HANDLER* ids.

  ```
  #define UM_HANDLE_INCREMENT    1
  ```

For both the *Process management system calls* and *File management system calls*, besides the effective implementation of all the system calls defined in *syscall_func.h*, we also need to implement functions for working with the previously specified handler, namely:

- *SyscallPreinit.* A function that initializes the *m_syscallData* presented previously by creating both the list and the lock. The function needs to be called only once from the *SystemPreinit* function defined in *system.c*.

  ```
  void
  _No_competing_thread_
  SyscallPreinit(
      void
  );
  ```

- *_SyscallGetNextUmHandleId.* Function which retrieves a unique id for an *UM_HANDLER* (based on the implementation of get next thread|process id).

```
__forceinline
static
UM_HANDLE
_SyscallGetNextUmHandleId(
    void
);
```

- *InsertUmHandler*. Function used in order to insert an *UM_HANDLER* into the *UmHandlerList* defined in *_SYSCALL_DATA* synchronously.

```
static
void
InsertUmHandler(
        PUM_HANDLER PUmHandler
);
```

- *DeleteUmHandler*. Function used in order to delete an *UM_HANDLER* from the *UmHandlerList* defined in *_SYSCALL_DATA* synchronously. If the deleted handler is a process, delete all the *UM_HANDLERs* that have as a parent process the process that is being deleted (meaning all the children processes, threads and files). If the deleted handler is a thread, also delete all the *UM_HANDLERs* that represent files opened by this thread.

```
static
void
DeleteUmHandler(
        PUM_HANDLER PUmHandler
);
```

- *FindUmHandlerBasedOnUmHandle*. Function used in order to find an *UM_HANDLER* from the *UmHandlerList* defined in *_SYSCALL_DATA* synchronously based on the unique *UM_HANDLE* value received.

```
static
void
FindUmHandlerBasedOnUmHandle(
        UM_HANDLE UmHandle,
        OUT_PTR PUM_HANDLER* PUmHandler
);
```

- *FindUmHandlerBasedOnProcess.* Function used in order to find an *UM_HANDLER* from the *UmHandlerList* defined in *_SYSCALL_DATA* synchronously based on the process *PProcess* received.

```
static
void
FindUmHandlerBasedOnProcess(
        PPROCESS PProcess,
        OUT_PTR PUM_HANDLER* PUmHandler
);
```

- *FindUmHandlerBasedOnThread.* Function used in order to find an *UM_HANDLER* from the *UmHandlerList* defined in *_SYSCALL_DATA* synchronously based on the thread *PThread* received.

```
static
void
FindUmHandlerBasedOnThread(
        PTHREAD PThread,
        OUT_PTR PUM_HANDLER* PUmHandler
);
```

- *SyscallHandler.* The function needs to be updated in order to also deal with the system calls. More concretely, based on the *syscallId* present in the **R8** register, identify the system call that needs to be called and call it with the necessary parameters (defined from the Rbp register onwards).

3.2.2 Detailed Functionality

The detailed functionality of this assignment is:

1. **Program argument passing**

- The first step is the parsing of the *FullCommandLine* field that can be found inside the *Process* data structure. This field represent the program's arguments as a string, where the arguments are delimited by spaces. The parsing is done using **strchr**, where we split the string based on spaces and, in order to correctly populate the stack, we store them in an array in reverse order such that the first parameter of the stack (on top of the stack) is the last argument of the user process.
- Afterwards, the total stack size needs to be computed in order to know how much of the user space needs to be allocated in the virtual memory of the kernel (such that we can access the user space stack from inside the kernel). The steps taken in the computation of the total stack size are:
  - Compute the total length of the arguments (including the **NULL** terminators for each argument)
  - Increase the previously computed number in order to make it a multiple of 16.
  - Add the size of the addresses of the arguments to the total stack size (the number of arguments multiplied by 8; the size of a pointer in C).
  - Add two shadow spaces (two void pointers; 16 bits)
  - Add the size of the addresses where the address of the first argument of the process is located (one char** pointer; 8 bits)
  - Add the size of the number of arguments (QWORD; 8 bits)
  - Add a dummy return address (one void* pointer; 8 bits)
- After computing the total stack size, we call the *MmuGetSystemVirtualAddressForUserBuffer* in order to get the address in the kernel space of the user space stack. We will use this address in order to populate the stack in a top to bottom manner. Thus, we fill the stack as follows:
  - Copy the actual value of the arguments of the process (together with their **NULL** terminators) into the stack. Assure

that the first argument copied is actually the last argument of the process.
    - Align the stack to 16 bits.
    - Copy the addresses of the arguments of the process previously inserted. The addresses are the ones relative to the user space, not to the kernel space. Thus, we will populate the stack using the pointer to the kernel space memory but the addresses that will be added are from the user space memory.
    - Add two shadow spaces.
    - Add the address where the address of the first argument of the process is located. As previously mentioned, the address needs to be from the user-space memory.
    - Add the value which represent the total number of arguments.
    - Add a dummy return address.
  - After filling the stack, we need to make to call the *MmuFreeSystemVirtualAddressForUserBuffer* in order to free the memory from the kernel space.
  - Finally, we set the stack pointer.

2. **System calls for process management**

  - *SyscallProcessExit()*.
    - No parameter validation needed.
    - Find currently running process.
    - Based on the previously found process, find the UmHandler which represents this process.
    - Terminate the currently running process.
    - return the ExitStatus.
  - *SyscallProcessCreate()*.
    - Validate that all the input parameters are not null (Except for *Arguments*). Otherwise, return *STATUS_ UNSUCCESSFUL*.

  - For the received pointers, assert that they are inside the user space memory of the currently running process and that the rights (read/write) are correctly placed. In case they are buffers, validate these conditions for the entire length of the buffer. This can be done using the *MmuIsBufferValid()*. In our case, we need to check that the *ProcessPath* respects these conditions and, in case the *Arguments* parameter is not null, that it also respects them. It also needs to be checked for *ProcessHandle*. Otherwise, return *STATUS_ UNSUCCESSFUL*.
  - Map the pointers and buffers from the user space to kernel by using *MmuGetSystemVirtualAddressForUserBuffer()*
  - If the *ProcessPath* is relative, use *IomuGetSystemPartitionPath()* to create the full path.
  - Create the new process.
  - Create the *UmHandler* associated to this process and insert it into the *UmHandlerList*. *S*et the value of the *ProcessHandle*. *R*eturn the status of the call to *ProcessCreate()*.

- *SyscallProcessGetPid()*
  - Validate that none of the parameters are null. Otherwise, return *STATUS_ UNSUCCESSFUL*.
  - Assert that the *ProcessId* pointer is correctly defined (as previously presented). Otherwise, return *STATUS_ UNSUCCESSFUL*.
  - Map the pointers and buffers from the user space to kernel by using *MmuGetSystemVirtualAddressForUserBuffer()*
  - In case the *ProcesHandle* value is *UM_ INVALID_ HANDLE_ VALUE*, then return the PID of the currently running process.
  - Otherwise, find in the *UmHandlerList* the *UmHandler* based on the received *ProcessHandle*. If found, set the *ProcessId* value using this handler. Otherwise, return *STATUS_ UNSUCCESSFUL*.
  - Return *STATUS_ SUCCESSFUL*.

- *SyscallProcessWaitForTermination()*

    - Validate that none of the parameters are null. Otherwise, return *STATUS_ UNSUCCESSFUL*.
    - Assert that the *TerminationStatus* pointer is correctly defined (as previously presented). Otherwise, return *STATUS_ UNSUCCESSFUL*.
    - Map the pointers and buffers from the user space to kernel by using *MmuGetSystemVirtualAddressForUserBuffer()*
    - Based on the received *ProcessHandle*, find in the *UmHandlerList* the *UmHandle* associated with this handler. If found, call *ProcessWaitForTermination()* using the process associated with the found handler. Otherwise, if we do not find the handler associated with the process we are waiting for, we assume that the process has already finished its execution.
    - Return *STATUS_ SUCCESSFUL*.
  - *SyscallProcessCloseHandle()*.
    - Validate that the received parameter is not *UM_ INVALID_ HANDLE_ VALUE*.
    - Based on the received *ProcessHandle*, find in the *UmHandlerList* the *UmHandle* associated with this handler. If found, call *ProcessCloseHandle()* using the process associated with the found handler. Otherwise, return *STATUS_ UNSUCCESSFUL*.
    - Return *STATUS_ SUCCESSFUL*.

3. **System calls for thread management**
  - *SyscallThreadExit()*.
    - No parameter validation needed.
    - Find currently running thread.
    - Based on the previously found thread, find the UmHandler which is associated to this thread.
    - Terminate the currently running thread.
    - return the ExitStatus.
  - *SyscallThreadCreate()*.

  - Validate that all the input parameters are not null (except for *Context*). Otherwise, return *STATUS_ UNSUCCESSFUL.*
  - Assert that the parameters are correctly defined in the user space (for the *Context* parameter, only if it is not null). Otherwise, return *STATUS_ UNSUCCESSFUL.*
  - Map the pointers and buffers from the user space to kernel by using *MmuGetSystemVirtualAddressForUserBuffer()*
  - Create the new thread.
  - Create the *UmHandler* associated to this thread and insert it into the *UmHandlerList. S*et the value of the *ThreadHandle. R*eturn the status of the call to *ThreadCreate().*

- *SyscallProcessGetTid()*
  - Validate that none of the parameters are null. Otherwise, return *STATUS_ UNSUCCESSFUL.*
  - Assert that the *ThreadId* pointer is correctly defined (as previously presented). Otherwise, return *STATUS_ UNSUCCESSFUL.*
  - Map the pointers and buffers from the user space to kernel by using *MmuGetSystemVirtualAddressForUserBuffer()*
  - In case the *ThreadHandle* value is *UM_ INVALID_ HANDLE_ VALUE*, then return the TID of the currently running thread.
  - Otherwise, find in the *UmHandlerList* the *UmHandler* based on the received *ThreadHandle.* If found, set the *ThreadId* value using this handler. Otherwise, return *STATUS_ UNSUCCESSFUL.*
  - Return *STATUS_ SUCCESSFUL.*

- *SyscallThreadWaitForTermination()*
  - Validate that none of the parameters are null. Otherwise, return *STATUS_ UNSUCCESSFUL.*
  - Assert that the *TerminationStatus* pointer is correctly defined (as previously presented). Otherwise, return *STATUS_ UNSUCCESSFUL.*
  - Map the pointers and buffers from the user space to kernel by using *MmuGetSystemVirtualAddressForUserBuffer()*

    - Based on the received *ThreadHandle*, find in the *UmHandlerList* the *UmHandle* associated with this handler. If found, call *ThreadWaitForTermination()* using the thread associated with the found handler. Otherwise, if we do not find the handler associated with the thread we are waiting for, we assume that the thread has already finished its execution.
    - Return *STATUS_SUCCESSFUL*.
- *SyscallThreadCloseHandle()*.
    - Validate that the received parameter is not *UM_INVALID_HANDLE_VALUE*.
    - Based on the received *ThreadHandle*, find in the *UmHandlerList* the *UmHandle* associated with this handler. If found, call *ThreadCloseHandle()* using the thread associated with the found handler. Otherwise, return *STATUS_UNSUCCESSFUL*.
    - Return *STATUS_SUCCESSFUL*.

4. **System calls for file management**
    - *SyscallFileCreate()*
        - Validate that none of the parameters are NULL. Otherwise, return *STATUS_UNSUCCESSFUL*.
        - Validate that all the pointers are correctly defined as stated previously. Otherwise, return *STATUS_UNSUCCESSFUL*.
        - If the *Path* is relative, use *IomuGetSystemPartitionPath()* to create the full path.
        - Create the file using *IoCreateFile()*.
        - Create the *UmHandler* associated to this file.
        - Insert the previously created handler in the *UmHandlerList*.
        - Set the value of *FileHandle*.
        - Return the status of the call to *IoCreateFile()*.
    - *SyscallFileRead()*
        - Validate that none of the parameters are NULL. Otherwise, return *STATUS_UNSUCCESSFUL*.

  - Validate that all the pointers are correctly defined as stated previously. Otherwise, return *STATUS_ UNSUCCESSFUL*.
  - Based on the given *FileHandle*, find the *UmHandler* in the *UmHandlerList*. If not found, return *STATUS_ UNSUCCESSFUL*.
  - Using the *FileHandler* associated to the previously found *UmHandler*, call *IoReadFile()*.
  - Return the status of the call to *IoReadFile()*.

- *SyscallFileClose()*
  - Validate that the received parameter is not *UM_ INVALID_ HANDLE_ VALUE*.
  - Based on the received *FileHandle*, find in the *UmHandlerList* the *UmHandle* associated with this handler. If found, call *IoCloseFile()* using the file associated with the found handler. Otherwise, return *STATUS_ UNSUCCESSFUL*.
  - Delete the *UmHandler* previously found from the *UmHandlerList*.
  - Return the status of the call to *IoCloseFile()*.

3.2.3 Explanation of Your Design Decisions

Design decisions:

1. *Program argument passing*

   - We have made use of the *strchr()* function in order to parse the command line of the user program because it was easier to use. Nevertheless, the same functionality could have also been achieved using the *strtok_ s()* function.
   - The arguments are placed in the stack in the right order because we keep track of them in a reversed order. Thus, the last argument inserted on the stack (in a top to bottom manner) is in fact the first actual argument of the user process.
   - The overflowing of the stack is avoided by the fact that we actually compute the total needed stack size **before** filling the stack. By

doing this, we can check that the needed size is enough and it does not overflow the 8 memory pages allocated for the stack.

- Lastly, one important design decision is that, when working with pointers, you need to make sure that the arithmetical operations on pointers are done on the **byte** type, not on types that have a greater size than one byte. This can be assured by making use of the *PtrDiff()* function for subtracting from pointers and *PtrOffset()* function for adding to pointers.

2. *Syscalls*

    - For the implementation of system calls we decided to create a custom data structure which holds and Id and a type along with some other relevant information necessary to deal with different situations that the system calls need to address independent of the context. For generating the handler's Id, we make use of an incremental sequence, thus each handler Id is unique within the entire operating system.
    - In order to read/write data belonging to the user space into the kernel space, we first need to map that memory chunk from the user address space into the kernel address space. For this, we make use of the already predefined function called *MmuGetSystemVirtualAddressForUserBuffer()*.
    - We don't directly use *_VmIsKernelAddress()* to check whether a given address belongs to the kernel space. Instead, for validating the buffers received during system calls, we use MmuIsBufferValid(). For mapping memory from user address space into kernel address space we use the function *MmuGetSystemVirtualAddressForUserBuffer()*.
    - For implementing the *SyscallProcessWaitForTermination()* we make use of the kernel function *ProcessWaitForTermination()*. Each process contains an *EX_EVENT* for signaling the termination to the waiting processes. What *ProcessWaitForTermination()* does

is to add the process to the waiting list corresponding to the termination event of the process we are waiting for. When a process terminates, i.e. *ProcessTerminate()* function is called, the processes inside the event waiting list are signaled that the termination of the process they were waiting for has occurred.

- For handling any error that may appear and can cause the resources to not be released, we choose to implement the exception handler in the following manner: whenever an exception from a user application, we choose to close that process by calling *ProcessTerminate()* inside the Interrupt Service Routine. To check whether the exception came from the user application, *_IsrExceptionHandler()* makes use of the function *GdtIsSegmentPrivileged()*. Regarding the resources a process handles, when the process terminates, all the resources related to the system calls of the process are released (all the threads are stopped, all the files are closed, etc.)

- The cases in which a parent process waits for a children are the following:
  - the parent process calls *SyscallProcessWaitForTermination()* before the child terminated: that process will wait and will be signaled when the child terminates
  - the parent process calls *SyscallProcessWaitForTermination()* after the child terminated: when a process terminates, its corresponding handler is deleted from the handler list. Therefore, when a parent process tries to wait for a child for which the corresponding handler doesn't exist, we assume that the child has already terminated its execution, therefore the parent process will immediately return to its execution without waiting (with STATUS_SUCCESS)
  - when a parent process terminates before the child process, the child process will be terminated as well

3.3 Tests

The tests which make sure that our implementation of the assignment is correct have the following mechanism:

1. *Program argument passing.* The tests create a program with a variable number of arguments (none, one or more), check to see if the stack is properly created, read the arguments from the stack, print them and check if the final result is the same as the expected one.

2. *System calls - process*

   - Test to see if the processes are closed/terminated correctly given a normal execution and also given some edge cases, such as: closing the parent's handle or close a process twice
   - Test the process creation functionality by checking some common cases such as creating a single or multiple processes, creating processes with arguments. Additionally, some corner cases are tested i.e: create a process with bad pointer to memory access, creating a process with an invalid file path.
   - Test to see if a process successfully exits by calling SyscallProcessExit().
   - Check if the system call successfully return the process ID
   - Test if a process waits for a running/terminated process and some other corner cases: waiting for a process given a bad handler and wait for a process given a closed handle

3. *System calls - threads*

   - For this part, the same situations/methods are applied (the same from the process part)

4. *File handling*

- Test to see if a file is properly closed under normal conditions (by simply calling *SyscallFileClose*). Additionally, some other corner cases are checked: close a file given a bad handler, close STDOUT, close the same file twice
- Create a new file under normal conditions as well as testing edge cases when creating a file given: a bad pointer, an empty path, an already existing file, a null path. Additionally, creating the same file twice is also tested
- Test to see if a valid file is read. The read file system call is also tested given a bad handler, a bad pointer. Lastly, it's also tested if one can read from STDOUT, read zero characters, and read in a buffer belonging to the kernel address space.

cd C:-Devel

Chapter 4

Design of Module *virtualmemory*

4.1 Assignment Requirements

4.1.1 Initial Functionality

HAL9000 currently works with Virtual Memory and supports allocation and deallocation operations, but it doesn't provide dedicated system calls for user applications in this regard. Having this in mind, we need to include two new system calls which a user application can use in order to take advantage of the Virtual Memory mechanism: *SyscallVirtualAlloc()* and *SyscallVirtualFree()*.

Moreover, the current implementation of HAL9000 currently has a fixed size stack (8 pages). To make the management more efficient, we need to implement a variable sized stack which applies dynamic allocation in case of stack overflow.

Also, the current implementation allows a process to occupy an unlimited number of physical address pages. Thus, in order to eliminate this, a swapping mechanism needs to be implemented that will limit the number of physically mapped pages a process has by making use of a swap file. The limit of pages will be imposed using process quotas.

Furthermore, the current implementation allows a process to have any number of files opened or physically mapped frames at any point in time. We need to limit this behavior. The process quotas part will be addressed (since it is used in the swapping mechanism) during the swap presentation.

Lastly, when it comes to virtual memory allocation, a user application should have the possibility of allocating pages of virtual memory directly initialized to 0, feature which is currently not supported by HAL9000.

4.1.2 Requirements

The requirements of the “Virtual Memory” assignment are the following:

- *Per Process Quotas.* The number of physically mapped frames or the number of open files a process can have needs to be limited to a predefined constant.
- *Zero Pages.* We have to implement a mechanism for allocating virtual address pages filled with zero bytes. zero bytes.
- *System calls.* We have to implement two new system calls in order to support the allocation and deallocation of virtual memory for user applications, namely *SyscallVirtualAlloc()* and *SyscallVirtualFree()*.
- *Stack Growth.* Implement a dynamically allocated stack which starts with a small number of pages allocated and increases on demand.
- *Swapping.* A swapping mechanism needs to be implemented in order to limit the number of physically mapped pages a process can have.

4.1.3 Basic Use Cases

For the **virtual memory syscalls**, one can distinguish the following use cases:

- Allocate a certain amount of virtual memory for a user program using *SyscallVirtualAlloc()*
- Free a previously allocated virtual memory region for a user program using *SyscallVirtualFree()*

For the **stack growth** part, the following use cases can be identified:

- Starting with a stack size of 1 page, allocate additional pages as the user program requires more memory for the stack until a certain threshold is reached

For the **process quotas** part, the following use cases can be identified:

- If a process opens too many files, that file will not be opened until the process closes an already opened file.
- If a process has too many physically mapped frames, the swapping mechanism will ensure that the additional frames will be moved onto the swap file.

For the **swapping** part, the following use cases can be identified:

- When a process exceeds its maximum number of pages it can have physically mapped at a moment in time, the swapping mechanism needs to decide using a heuristic what page of the current process has to be swapped out.

For the **zero pages** part, the following use cases can be identified:

- A user application can request a region of virtual memory directly filled with zero bytes.

4.2 Design Description

4.2.1 Needed Data Structures and Functions

4.2.1.1 Virtual Memory Syscalls

For correctly implementing syscalls for virtual memory, the following data structure has been added:

```
typedef struct _VMM_ADDR_RESOURCE
{
        // start of vmm allocation
        PVOID Address;
        // size of vmm allocation
        QWORD Size;

} VMM_ADDR_RESOURCE, * PVMM_ADDR_RESOURCE;
```

The role of this data structure is to store the address and the size of all virtual memory allocations made through *SyscallVirtualAlloc()*. This is needed to ensure that the user application doesn't free virtual memory that was not allocated using *SyscallVirtualAlloc()*.

To accomodate the addition of this new data structure, the following data structures from the previous assignment were enhanced:

```
typedef union _UmHandlerResource {
        // the created process
        // (in case the UmHandlerType is _UM_HANDLER_PROCESS )
        PPROCESS PProcess;

        // the created thread
        // (in case the UmHandlerType is _UM_HANDLER_THREAD )
        PTHREAD PThread;

        // the created / opened file
        // (in case the UmHandlerType is _UM_HANDLER_FILE )
        PFILE_OBJECT FileHandler;

        // the virtual memory allocated
        VMM_ADDR_RESOURCE VmmResource;

} UmHandlerResource;


// UM_HANDLER enum specifying the type of the UM_HANDLER resource
typedef enum _UM_HANDLER_TYPE
{
        _UM_HANDLER_THREAD,
        _UM_HANDLER_PROCESS,
        _UM_HANDLER_FILE,
        _UM_HANDLER_VMM_RESOURCE

} UM_HANDLER_TYPE;
```

First, the new data structure has to be added in the *_UmHandlerResource* union to be able to use it inside *UM_HANDLER* structure.

Secondly, the enum containing the type of the handler has to be modified so that we can correctly identify a handler of type *_UM_HANDLER_VMM_RESOURCE*.

Additionally, in order to validate the de allocation of a virtual memory zone, *VmFindReservation()* is used as a buffer check. To make this function and the corresponding data structure visible (*VMM_RESERVATION*) in other files, we needed to move the data structure from and function signature from **vm_reservation_space.c** to **vm_reservation_space.h**.

4.2.1.2 Stack growth

For the stack growth part, not many changes were made. We only needed to add a new constant for the default stack size, i.e. **UM_STACK_DEFAULT_SIZE** as well as a new field in the thread structure for keeping track of the size of each thread:

```
typedef struct _THREAD
{
    ...
        // User stack size (in pages)
        WORD                                        UserStackSize;
    ...
} THREAD, * PTHREAD;
```

The field *UserStackSize* is initialized to *UM_STACK_DEFAULT_SIZE* (1 page).

4.2.1.3 Process Quotas

For the process quotas part, the following data structure needs to be modified:

1. *PROCESS* data structure needs to me enhanced in order to keep track of the number of opened files.

   ```
   typedef struct _PROCESS
   {
         ...
   // the count of open files
   QWORD        OpenFileCount;
         ...
   } PROCESS, *PPROCESS;
   ```

2. we will also need to create a constant that represents the maximum number of open files a process can have. This constant is called *PROCESS_MAXIMUM_OPEN_FILES* and has a value of **16**.

The following functions need to be modified:

1. *SyscallFileCreate* needs to be modified in order to create the file only if the process has not exceeded the maximum number of files it can create. Moreover, if it can create the file, the process' count needs to be increment with **1**.

2. *SyscallFileClose* needs to be modified in order to also decrement the process' count when the file is closed.

4.2.1.4 Swapping

For the swapping part, the following data structures need to be added or modified:

1. *IOMU_DATA* data structure needs to be enhanced in order to keep a bitmap that represents the state of the swap file.

```
typedef struct _IOMU_DATA
{
...
// bitmap state of the swap file
BITMAP                  SwapBitmap;

// lock to ensure exclusive access
LOCK                    SwapBitmapLock;

// bitmap data
PVOID                  SwapBitmapData;

// the size of the swap file
QWORD                  SwapFileSize;
...
} IOMU_DATA, *PIOMU_DATA;
```

2. *FRAME_ MAPPING* data structure needs to be created in order to be able to keep track of the mappings that have been done with respect to the swap file.

```
typedef struct _FRAME_MAPPING {

// the virtual address of the page
PVOID                                   VirtualAddress;

// the physical address of the page
// (in case it is not in the swap file)
PHYSICAL_ADDRESS    PhysicalAddress;

// the index offset at which the page
// is in the swap file (in case it is present there)
DWORD                                   SwapIndex;

// a flag denoting whether the page is in the swap file
// or in the physical memory
BOOLEAN                                 InSwapFile;

// the page rights of the current page
PAGE_RIGHTS                    PageRights;

// the variable that will be the basis of
// the swapping heuristic
QWORD                  AccessCount;

// the entry that will be used in the
// FrameMappingsList in the process
LIST_ENTRY                     FrameMappingEntry;
} FRAME_MAPPING, *PFRAME_MAPPING;
```

3. *PROCESS* data structure needs to be enhanced in order to keep track of its frame mappings.

```
typedef struct _PROCESS
{
    ...
// the list of mappings
LIST_ENTRY                     FrameMappingList;

// lock to protect the list of mappings
LOCK                           FrameMappingLock;

// number of physically mapped pages
QWORD                          NumberOfMappedPhysicalFrames;
    ...
} PROCESS, *PPROCESS;
```

4. Also, a new constant needs to be added in order to ensure an upper limit on the number of physical frames a process can have at a point in time. This constant is called *PROCESS_MAXIMUM_PHYSICAL_FRAMES* with the value of **16**.

The following functions have to be created or modified:

1. *_IomuInitializeSwapFile()* needs to be changed in order to also allow the computation of the *SwapSize* variable. Moreover, in this function we will also initialize the *SwapBitmap*, *SwapBitmapData* and *SwapBitmapLock*.

2. *IomuSwapOut()* needs to be defined in *iomu.c*. This function will allow swapping a page from the physical memory into the swap file.

```
STATUS
IomuSwapOut(
    IN      PVOID       VirtualAddress
);
```

3. *IomuSwapIn()* needs to be defined in *iomu.c*. This function will allow swapping a page from the swap file into the physical memory.

```
STATUS
IomuSwapIn(
    OUT     PVOID       VirtualAddress
);
```

4. *_VmmAddFrameMappings()* needs to be defined in *vmm.c*. This function will allow the creation of multiple frame mapping structures that will be added to the list of the current process.

```
static
void
_VmmAddFrameMappings(
    IN          PHYSICAL_ADDRESS    PhysicalAddress,
    IN          PVOID               VirtualAddress,
    IN          DWORD               FrameCount
);
```

5. *VmmSolvePageFault()* needs to be modified in order to add a frame mapping to the current process' list whenever a page fault is solved.

As one last addition, whenever the VMM tries to solve a page fault, it will firstly check if the respective virtual address is not already in the swap file. If it is, it will simply move the page out of the swap file and into the memory. Also, the *NumberOfMappedPhysicalFrames* of the current process needs to be incremented by **1** if the page fault has been solved. Nevertheless, the function *VmmCheckForSwapOutNeed()* needs to be called.

6. *VmmAllocRegionEx()* needs to be modified in order to add multiple frame mappings to the current process' list whenever an eager virtual allocation is performed. Also, the *NumberOfMappedPhysicalFrames* of the current process needs to be incremented with the number of eagerly allocated pages. Nevertheless, the function *VmmCheckForSwapOutNeed()* needs to be called.

7. *VmmFreeRegionEx()* needs to be modified in order to delete multiple frame mappings from the current process' list whenever a virtual address and its corresponding mappings are released. Also, the *NumberOfMappedPhysicalFrames* of the current process needs to be decremented with the number of pages that are unmapped by the *MmuUnmapMemoryEx()* function.

8. *VmmTick()* needs to be defined in *vmm.c*. This function will check, at each timer interrupt, all the processes' lists and will increment their *AccessCount* field based on the number of physically mapped pages that have been accessed since the last timer interrupt.

```
void
VmmTick(
    void
);
```

9. *ExSystemTimerTick()* needs to be modified in order to also call the *VmmTick()* function.

10. *VmmCheckForSwapOutNeed()* needs to be defined in *vmm.c*. This function will check, after each time a physical page has been mapped, if the current process exceeds the allowed maximum number of mapped pages. In that case, it will choose one of its pages to be swapped into the swap file. When the page is swapped, it will be deleted from the process. Intuitively, the physical memory will also be deallocated.

    ```
    void
    VmmCheckForSwapOutNeed(
        IN PVOID VirtualAddress
    );
    ```

4.2.1.5 Zero pages allocation

In order to correctly implement zero pages allocation, we first need a way to distinguish whether a page should be filled with zero bytes or not. This is the part where the *VMM_RESERVATION* structure comes in handy. More specifically, we chose to add a new field in this data structure, which tells us whether the memory region should be set to 0 or not:

```
    typedef struct _VMM_RESERVATION
{
        ...
        //specifies if the memory region should be filled with 0
        BOOLEAN     ZeroPageFlag;
        ...
} VMM_RESERVATION, * PVMM_RESERVATION;
```

4.2.2 Detailed Functionality

The detailed functionality for each part of this assignment is the following:

1. **System calls for virtual allocation**

    - *SyscallVirtualAlloc()*:

  - validate the corresponding page rights (only read and write); return *STATUS_ UNSUCCESSFUL* otherwise
  - check that the base address pointer is not null and has the correct access rights;
  - check that the allocated address pointer is not null and validate the right
  - allocate the required amount of virtual memory using VmmAllocRegionEx()
  - create the handler corresponding to this virtual allocation (base address and size)
  - return *STATUS_ SUCCESS*

- *SyscallVirtualFree()*:
  - validate the given parameters: starting address of the virtual memory, free type, and size
  - check if the provided virtual address is present in the reservation structure using VmFindReservation;
  - return *STATUS_ UNSUCCESSFUL* if not
  - find the corresponding *UM_ HANDLER* for this virtual allocation to make sure that the memory was allocated using *SyscallVirtualAlloc()*; return *STATUS_ UNSUCCESSFUL* if handler not found
  - free the virtual memory area using *VmmFreeRegionEx()*
  - return *STATUS_ SUCCESS*

2. **Stack growth**

    - detect a stack page fault exception: check if the faulting address read from CR2 is the same as the current stack pointer retrieved from RSP register: if so, a stack page fault exception was generated
    - check if the thread's user stack size is smaller than *STACK_MAXIMUM_SIZE* (16 pages)
    - if there is still room for allocating pages to the stack, allocate one more page
    - increase the number of pages of the thread's stack by 1 as well as the number of page faults

3. **Process Quotas**

    - *SyscallFileCreate()*:
        - Now, the function needs to firstly check that the process has not exceeded the number of open files it has.
        - In case it can no longer open files, an unsuccessful status will be returned.
        - In case it can open the file, then the process' count will be incremented.
    - *SyscallFileClose()*:
        - The function will need, besides its original implementation, to also decrement the process' count.

4. **Swapping**

    - *SwapOut()*:
        - The first is to identify a free space in the swap file. This can be done by using the *BitmapScan()* function by specifying that we are looking for a **single unset** bit.

  - After identifying the free space, we will identify the frame mapping from the current process that corresponds to the given virtual address. This frame will be updated by setting the swap flag and the swap index. Nevertheless, *AccessCount()* to **0**.
  - The next step is to copy the contents of the current page into the swap file. We will use the *IoWriteFile()* function, where we will specify the offset based on the index received from the search on the bitmap.
  - Last but not least, the current page needs to be released from the physical memory. In this case, the function *VmmFreeRegionEx()* will be used.

- *SwapIn()*:
  - The first step is to identify the frame mapping corresponding to the given virtual address.
  - Based on the swap index found, read the content of the page from the file using *IoReadFile()* and save it into a buffer.
  - For the current process, allocate a new page eagerly using *VmmAllocRegionEx()*.
  - Copy the page from the buffer into the allocated physical memory that is returned by the previously specified function.
- *VmmCheckForSwapOutNeed()*:
  - The first step is to check if the process has exceeded the maximum number of physically mapped frames it can have.
  - In that case, the next step is to iterate through the mappings of the current process and look for the mapping that is physically present and has the minimum *AccessCount* value.
  - For this mapping, we will call the *SwapOut()* function.
- *VmmTick()*:
  - The first step of this function is to iterate through all the running processes in the system.

  - For each process, we will iterate through each mapping.
  - For each mapping, we will look for the *access bit* and *dirty bit*. If either of them is **NON-NULL**, then the page has been accessed since the last timer interrupt. In that case, the *AccessCount* field will be incremented.
- *ExSystemTimerTick()*:
  - Will call *VmmTick()* at each timer interrupt.
- *_VmmAddFrameMappings()*:
  - Based on the virtual memory allocation, meaning the starting virtual address, starting physical address and number of frames, this function will create multiple mapping structures (as many as the number of frames) that will be added to the list of the current process.
- *VmmAllocRegionEx()*:
  - In case the allocation has been performed eagerly, then the *_VmmAddFrameMappings()* will be called. This needs to be done only if the allocation is **not done** by the stack. This will be checked by using a flag.
  - The current process' *NumberOfMappedPhysicalFrames* is incremented with the number of frames allocated.
  - After creating the mapping structures, the function *VmmCheckForSwapOutNeed()* is called.
- *VmmSolvePageFault()*:
  - The first step is to check if the needed address is not in the swap file. In case it is not, then the normal flow is performed.
  - Otherwise, the corresponding frame needs to be taken out of the swap file using *SwapIn()*.
  - At the end of the function, if the page fault has been solved, the current process' *NumberOfMappedPhysicalFrames* is incremented with **1**. Moreover, the function *VmmCheckForSwapOutNeed()* will be called.

- *VmmFreeRegionEx()*
  - Will delete the mappings from the current process' list.
  - Will decrease the *NumberOfMappedPhysicalFrames* field by the number of frames that *MmuUnmapMemoryEx()* function unmaps.
  - Afterwards, it will perform its initial flow.

5. **Zero Pages**

   - the first step would be to simply call *SyscallVirtualAlloc()* previously discussed using **VMM_ALLOC_TYPE_ZERO** allocation type
   - inside *VmmAllocRegionEx()* the allocation type will be checked and in the case of a zero page allocation, the previously defined flag from *VMM_RESERVATION* structure will be set, i.e. *ZeroPageFlag*
   - in the case of a page fault, the *ZeroPageFlag* from *VMM_RESERVATION* is retrieved and checked
   - if the flag is set, then the memory mapped as a result of handling the page fault will be also filled with zero bytes using the function *memzero()*

4.2.3 Explanation of Your Design Decisions

1. *System calls for virtual allocation*

   - We added a new data structure for keeping track of the virtual memory regions allocated using SyscallVirtualAlloc(). This data structure was added in the form of a *UM_HANDLER* created each time a new allocation was performed.
   - When a deallocation of a virtual memory region is requested, VmFindReservation() is used in order to check whether the provided address is a valid one (address corresponding to the starting point of a virtual memory region)

2. *Stack growth*

    - Modified the default size for a newly created stack (1 page instead of 8)
    - Added a new field in the thread structure for keeping track of the number of pages allocated to the current thread (initialized to 1). This addition was needed in order to make sure that the stack doesn't exceed the maximum size imposed by STACK_MAXIMUM_SIZE (16 pages)
    - To detect whether a stack page fault occurred we check if the faulting address read from CR2 is the same as the current stack pointer retrieved from the RSP register: if so, a stack page fault exception was generated

3. *Process Quotas*

    - We have chosen to return an unsuccessful status when failing to open a file instead of randomly closing another file because we believe that it's the process' duty to perform this operation.
    - Regarding the physically mapped frames, we have chosen to treat the problem when the exceed occurs by heuristically moving a frame into the swap file.

4. *Swapping*

    - The identification of the frame that needs to be swapped is done using the frame mapping structure that is kept inside the process as a list.
    - In order to perform the eviction of a frame when none is available, we chose to eliminate the frame that has been the least accessed amongst the other frames. This heuristic is provided by the *AccessCount* field present in the mapping structure inside the process, which represents the total number of accesses this page has since it has been mapped in the physical memory. We can

identify if a page has been accessed if any of the two flags, *dirty* and *accessed*, are NON-NULL. The check is performed based on the timer interrupt.

- The functionality we have provided is built upon the existing allocation and page fault solving. Because of this, the race conditions are already handled by the initial implementation.

- When performing the swapping, the page is deallocated from the initial process holder. The only place inside the process structure where we have a link to this page is in the mapping structure we have provided. Because of this, that means that the initial process does no longer have any connection to the frame that was initially used to hold the page.

- We have chosen to perform the frame eviction only on the frames of the current process. The reason for this is that, in our opinion, when performing the swap, that action is performed **only** because that process has exceeded the maximum number of pages it is allowed to have physically mapped. Because of this, there is no point in evicting the frame of another process.

- Because of the way we have chosen to implement the swapping mechanism, any time a process tries to access a page that is held inside the swap file, a page-fault will occur. Thus, we make use of this page-fault exception to bring back the page from the swap file and map it again to the process.

5. Zero pages

   - In order to keep track of the pages that should be initialized with zero, we chose to add an additional flag inside *VMM_ RESERVATION*, i.e. *ZeroPageFlag*, which is used to fill the newly mapped paged from the physical memory with zero bytes when a page fault exception is treated.

4.3 Tests

The implementation of each requirement is tested using the following cases:

1. *Process Quotas*

   - Open the maximum number of allowed files, after that, close half of them and try to open new ones. This test represents the most common use case.
   - Try to open the maximum number of allowed files. Check to see if it is possible
   - Try to open the more handlers than possible possible, at least 16 must be correctly open, and the rest should fail.

2. *Stack growth*

   - The first test calls a recursive function for *NO_OF_TIMES_TO_CALL_RECURSIVE_FUNCTIONS* (10000), this calls being allocated in the stack. The stack must extend its size such that it can allow this test to pass.
   - The test calls *_AllocateAlmostAPageOfLocalVariables* for a certain number of times. It which places the recursive call and a local variable on the stack.

3. *System calls for virtual allocation*

   - Tested if one can write in a **READ-ONLY** allocated page.
   - Eagerly allocate a huge memory area and write into it.
   - Lazily allocate a huge memory area and write into it.
   - Allocate a page of virtual memory and write into it.
   - Try to allocate a page with *write* and *execute* rights (this should not be allowed).
   - Try to free a invalid virtual memory address.

- Try to free a bigger memory space than the one allocated.
- Try to lazily allocate 2GB of zero pages and tries to free it .

4. *Swapping*

   - Allocates 16Mb of virtual memory and, afterwards, goes over each page and writes into it. Afterwards, it reads the content and checks if the content is consistent.
   - Allocates 256Mb of virtual memory using zero pages and, afterwards, goes over each page and reads its content and checks if the values read are all 0.
   - Allocates 16Mb of virtual memory using zero pages and, afterwards, goes over each page and writes into it. Afterwards, it reads the content and checks if the content is consistent.