# The Design of *HAL9000* Operating System

Ardelean Octavian

11 mai 2021

## Rezumat

The aim of this project is to implement various functionalities of the HAL9000 operating system.

# 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 BBB
   (c) VM: dealt with CCC
2. Egri Julia
   (a) Threads: dealt with Priority Scheduler.
   (b) Userprog: dealt with BBB
   (c) VM: dealt with CCC
3. Ardelean Octavian
   (a) Threads: dealt with Priority Donation.
   (b) Userprog: dealt with BBB
   (c) VM: dealt with CCC

# Capitolul 2

# The (One) Way to Proceed for Getting a Reasonable Design

## 2.1 General Considerations

There are multiple strategies to develop a software application, though basically all of them comprise the following four phases:

1. establish the *application requirements* or specification;

2. derive the ways the requirements can be realized / satisfied, i.e. *designing* the application;

3. *implement* the design, e.g. write the corresponding code;

4. check if the implementation satisfies the specification, at least by *testing* (as exhaustive as possible) the developed application.

In practice, a perfect and complete design is not entirely possible from the beginning for most of the projects. So, at least the last three phases actually correspond to a progressive and repeating process, i.e. make a first design, implement it, test the resulting code, see what is working bad or missing functionality, go back and change the design, make the corresponding code changes or additions, test them again and so on.

I want, however, to warn you that even if we cannot make a perfect design from the beginning that does not mean that we do not have to make any design at all and just start writing code. This is a really bad idea. And actually, in my opinion, when you start writing code without a more or less formal design, what you actually have to do is to derive an on-the-fly design. What I mean is that you cannot just write "some" code there, hoping to get the required functionality. You must think of *how* to code and *what* code to write and this is basically a (hopefully, logical) plan, i.e. a design. Such a strategy, however, results most of the time and for most of the people in just poorly improvisation and requires many returns to the "design" phase to change data structures and code. In short, a lot of lost time and bad results.

Coming back to the idea that we cannot make a complete design from the beginning, there are a few ways to understand this and reasons of having it. Firstly, it is generally difficult to cover all the particular cases, especially for very complex systems and requirements. That means that what you get first is a sort of a general design, establishing the main components of your application and their basic interrelationships. It is not surely that you immediately could start writing code for such a design, but it is very possible to be able to write some prototype, just to see if your ideas and design components could be linked together. On way or another, the next major step is to go deeper for a more detailed design. Secondly, one reason of not getting a complete design from the beginning is just because you want to concentrate on a particular component firstly, and only than to cope with the others. However, this is just a particular case of the first strategy, because it is not possible to deal with one application component without knowing firstly which are the others and how they depend on one another. Thirdly, maybe it is not needed to get a complete detailed design from the beginning, just because the application components are dealt with by different teams or, like in your case, different team members. It is not needed in such a case to deal with the complexity of each application component from the beginning, as each one be will be addressed latter by its allocated team (members), but just try to establish as precise as possible, which are the application components and how they need to interact each other. In your *HAL9000* project the application components are most of the time already established, so what remains for you is only to clarify the interactions and interfaces between them. After such a general design, each team (member) can get independently into a more detailed design of his/her allocated application component. In conclusion, you need to derive at least a general design before starting writing any code and refine that design later.

Take into account, however, that in or project we have distinct deadlines for both design and implementation phases, and that you will be graded for the two relatively independent (thus, design regrading will be done only occasionally). This means that you have to try to derive a very good and as detailed as possible design from the beginning.

Another practical idea regarding the application development phases is that there is no clear separation between those phases and especially between the design and implementation ones. This means that during what we call the design phase we have to decide on some implementation aspects and, similarly, when we are writing code we still have to take some decisions when more implementation alternatives exists (which could influence some of the application non-functional characteristics, like performance) or some unanticipated problems arise. Even taking into account such realities, in this design document we are mainly interested (and so you have to focus) mainly on the design aspects. But, as I said above, I will not expect you providing a perfect design, which would need no changes during its implementation. However, this does not mean that you are free to come with an unrealistic, incoherent, illogical, hasty, superficial design, which does not deal with all the (clear and obvious) given requirements.

One important thing to keep in mind when you make your design and write your design document is that another team member has to be able to figure out easily what you meant in your design document and implement your design without being forced to take any additional design decisions during its implementation or asking you for clarifications. It is at least your teacher you have to think of when writing your design document, because s/he has to understand what you meant when s/he will be reading your document. Take care that you will be graded for your design document with approximately the same weight as for your implementation.

Beside the fact that we do not require you a perfect design, and correspondingly we do not grade with the maximum value only perfect designs (yet, please, take care and see again above what I mean by an imperfect design), your design document must also not be a formal document. At the minimum it should be clear and logical and complies the given structure, but otherwise you are free to write it any way you feel comfortable. For example, if you think it helps someone better understand what you mean or helps you better explain your ideas, you are free to make informal hand-made figures, schemes, diagrams, make a photo of them or scan them and insert them in your document. Also, when you want to describe an algorithm or a functionality, you are free to describe it any way it is simpler for you, like for instance as a pseudo-code, or as a numbered list of steps. However, what I generally consider a bad idea is to describe an algorithm as an unstructured text. On one hand, this is difficult to follow and, on the other hand, text could generally be given different interpretations, though as I already mentioned your design should be clear and give no way for wrong interpretation, otherwise it is a bad design.

Regarding the fact that design and implementation could not be clearly and completely separated, this is even more complicated in your *HAL9000* project that you start from a given code of an already functional system. In other words you start with an already partially designed system, which you cannot ignore. This means, on one hand, that you could be restricted in many ways by the existing design and *HAL9000* structure and, on the other hand, that you cannot make your design ignoring the *HAL9000*' code. Even if, theoretically, a design could be abstract enough to support different implementations (consequently, containing no particular code), your *HAL9000* design has to make direct references to some existing data structures and functions, when they are needed for the functionality of the *HAL9000* component your are designing. So, do not let your design be too vague (abstract) in regarding to the functionality directly relying on exiting data structures and functions and let it mention them explicitly. For instance, when you need to keep some information about a thread, you can mention that such information will be stored as additional fields of the "*THREAD*" data structure. Or, when you need to keep a list of some elements, you have to use the lists already provided by *HAL9000* and show that by declaring and defining your list in the way *HAL9000* does, like below:

```
// this is the way HAL9000 declares a generic list
LIST_ENTRY myCoolList;

typedef struct _MY_LIST_ELEM {
        ... some fields ...

        // this is needed for linking MY_LIST_ELEM in the
        // myCoolList list
        LIST_ENTRY ListElem;
} MY_LIST_ELEM, *PMY_LIST_ELEM;
```

The next sections illustrate the design document structure we require and describe what each section should refer to.

## 2.2 Application Requirements. Project Specification

### 2.2.1 "What you have" and "What you have to do"

In this section you have to make clear what you are required to do for each particular assignment of the *HAL9000* project. In the *HAL9000* project, you are already given the assignment requirements, so it is not your job to establish them. You must, however, be sure that you clearly understand them. Having no clear idea about what you have to do, gives you a little chance to really get it working. Please, do not hesitate to ask as many questions about such aspects on the *HAL9000* forum (on the moodle page of the course) as you need to make all the requirements clear to you. Take care, however, that you have to do this (long enough) before the design deadline, such that to get an answer in time. We will do our best in answering your questions as fast as possible, though we cannot assure you for a certain (maximum) reaction time. You will not excused at all if you say you had not understood some requirements when you will be presenting your design document.

So, for this small section, take a moment and think of and briefly write about:

1. what you are starting from, i.e. what you are given in terms of *HAL9000* existing functionality, and

2. what you are required to do.

### 2.2.2 "How it would be used"

Making clear the requirements could be helped by figuring some ways the required functionality would be used once implemented. For this you have to describe briefly a few common use-cases, which could later be used as some of the implementation tests.

You could use for this the tests provided with the *HAL9000* code in the "tests/" subdirectory. Take at least a short look at each test case to identify common cases.

## 2.3 Derive the Application's Design

Generally, you have to follow a top-down design approach, starting with a particular requirement and identifying the inputs it generates to your application (i.e. *HAL9000* OS). Such that you could establish the "entry (starting) points" in your system to start your design from.

Next, you also have to identify the logical objects (i.e. data structures, local and global variables etc.) implied and affected by the analyzed requirement and the operations needed to be performed on them. Also establish if you need to introduce and use additional information (i.e. fields, variables) in order to make such operations possible.

Once you established the information you need to keep in order to dealt with the analyzed requirement, you could decide on the way to keep track of and manage that data. In other words, this is the way you can *identify the needed data structures* and *operations on them*. There could not necessarily be just one solution. For example, at one moment you could use a linked list or a hash table. In order to decide for one or another, you have to figure out which one helps you more, which one fits better for the most frequent operation and other things like these. Once you decided on the data structures, you have to establish where and how they are (1) *initialized*, (2) *used*, (3) *changed*, and finally (4) *removed*.

This way you could identify the places (e.g. other system components, functions) you have to add the needed functionality. In terms of *HAL9000* code you could identify the functions needed to be used, changed or even added.

As a result of your requirement analysis you could organize your resulting design like below.

### 2.3.1 Needed Data Structures and Functions

Describe the *data structures* and *functions* (only their signature and overall functionality) needed by your design and what they are used for. Describe in few words the purpose of new added data structures, fields and functions. As mentioned before, cannot ignore the fact that *HAL9000* is written in C. Thus, describe you data structures in the C syntax referring, when the case, to existing data structures, like in the example given above. If you only need to add new fields in existing data structures, mention only the added fields, not all of them.

```
typedef struct _EXISTENT_DATA_STRUCTURE {
    ...
    int NewField;
    char NewField2;
    ...
} EXISTENT_DATA_STRUCTURE, *PEXISTENT_DATA_STRUCTURE;

typedef struct _NEW_DATA_STRUCTURE {
    ... new fields ...
} NEW_DATA_STRUCTURE, *PNEW_DATA_STRUCTURE;

STATUS NewFunction(void);
```

### 2.3.2 Detailed Functionality

This should be **the largest and most important section** of you design document. It must describe **explicitly**, in words and pseudo-code the way your solution works. DO NOT INCLUDE CODE HERE. This is a design document, not a code description one. As much as possible try to describe the design principles, not implementation details.

Give examples, if you think they can make your explanation clearer. You are free to use any other techniques (e.g. use-case diagrams, sequence diagrams etc.) that you think can make your explanation clearer. See Figure 5.2 below to see the way images are inserted in a Latex file.

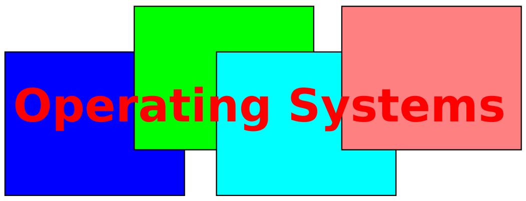


Figura 2.1: Sample image

Here you have a pseudo-code description of an algorithm taken from http://en.wikibooks.org/wiki/LaTeX. It uses the *algpseudocode* package. Alternatively, you can use any other package and environment of same sort you like.

```
if $i \geq maxval$ then
    $i \leftarrow 0$
else
```

```
        **if** $i + k \leq maxval$ **then**
            $i \leftarrow i + k$
        **end if**
    **end if**
```

### 2.3.3 Explanation of Your Design Decisions

Justify very briefly your design decisions, specifying other possible design alternatives, their advantages and disadvantages and mention the reasons of your choice. For instance, you can say that you decided for a simpler and easier to implement alternative, just because you had no enough time to invest in a more complex one. Or just because you felt it would be enough for what *HAL9000* tests would check for. This could be viewed as a pragmatical approach and it is not necessarily a bad one, on the contrary, could be very effective in practice.

## 2.4 Testing Your Implementation

Please note that the *HAL9000* code is provided with a set of tests that are used to check and evaluate your implementation. The *HAL9000* tests could be found in the "tests/" subdirectory, organized in different subdirectories for each different assignments (like, "threads", "userprog" etc.).

To find out the names of all the tests to be run for a module you can build the *RunTests* project and wait for execution to finish.

Actually, this is the first command that will be executed on your implementation when graded, so please, do not hesitate do run it by yourself as many times as needed during your *HAL9000* development, starting from the design phase.

In this section you have to describe briefly each of the given *HAL9000* tests that will be run in order to check the completeness and correctness of your implementation. Take care that your grade is directly dependent on how many tests your implementation will pass, so take time to see if your design take into account all particular usage scenarios generated by all *HAL9000* tests.

## 2.5 Observations

You can use this section to mention other things not mentioned in the other sections.

You can (realistically and objectively) indicate and evaluate, for instance:

- the most difficult parts of your assignment and the reasons you think they were so;
- the difficulty level of the assignment and if the allocated time was enough or not;
- particular actions or hints you think we should do for or give to students to help them better dealing with the assignments.

You can also take a minute to think what your achieved experience is after finishing your design and try to share that experience with the others.

You can also make suggestions for your teacher, relative to the way s/he can assist more effectively her/his students.

If you have nothing to say here, please remove it.

# Capitolul 3

# Design of Module *Threads*

## 3.1 Assignment Requirement

In all cases when multiple threads are waiting for the same lock, they will be put in order of priority in the waitingList which will then be processed in FIFO manner.

### 3.1.1 Requirements

*Priority Donation.* As described in the *HAL9000* documentation after the fixed priority scheduler is implemented a priority inversion problem occurs. Namely, threads with medium priority may get CPU time before high priority threads which are waiting for resources owned by low priority threads. You must implement a priority donation mechanism so the low priority thread receives the priority of the highest priority thread waiting on the resources it owns.

We have multiple cases for priority inversion and we need to treat each of them individually: change lock implementation, cases of multiple-locks, case of nested donation.

### 3.1.2 Basic Use Cases

1. Use case 1: 2 Threads fighting for the same lock - fig 3.1

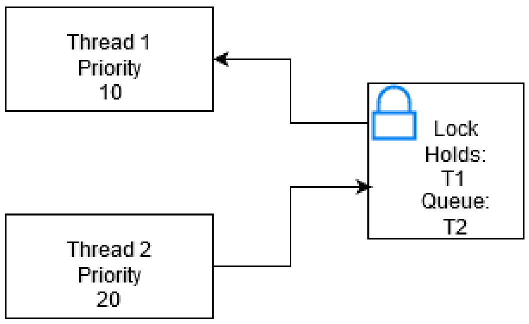


Figura 3.1: Priority donation: case 1

   We consider two threads: T1, T2 and a lock. T1, with a priority of 10, holds the lock (making it it's owner) when T2, with a priority of 20, will request the lock. Since T1 is it's holder, the LockTryAcquire() function will return false and enter the waiting list. When the system realises a thread with a higher priority waits for a thread with a lower priority, priority donation will occur: T2 will temporarily donate it's priority to T1 such that both threads have a priority of 20. Because the thread scheduling will work based on a FIFO principle, T1 will finish first and set it's priority back to it's original.Next, T2 leaves the queue and acquires the lock and sets itself as the lock holder.

2. Use case 2: 3 Threads fighting for 2 locks -fig 3.2

   Similar to the first case, but we will need to apply multiple priority donations. Thread 3 has the highest priority and will try to access lock 2. Since the lock has t1 as it's holder, T3 will donate it's priority to t1. After t1's execution, T3 will become the new holder and remove itslef from the waitinglist. The same witll happen then with T2.

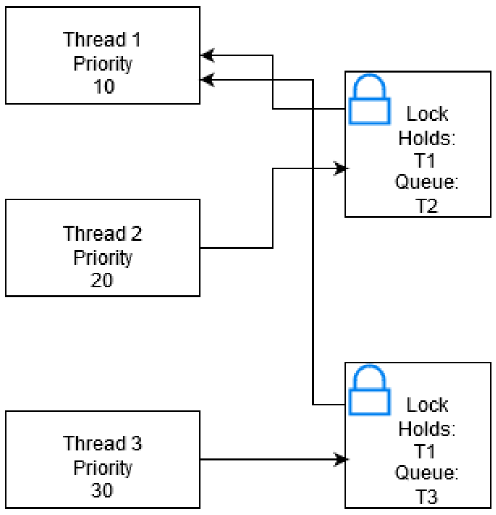


Figura 3.2: Priority donation: case 2

## 3.2 Design Description

### 3.2.1 Needed Data Structures and Functions

We will use the following data structures and functions: ThreadSystemData, MutexInit, MutexAcquire, MutexRelease, Lock, LockInit, LockAcquire, LockTryAcquire, LockIsOwner, LockRelease, Thread, ThreadState, ThreadSetPriority, ThreadBlock, ThreadUnblock, ThreadTerminate, ThreadTakeBlockLock, SetCurrentThread.

### 3.2.2 Detailed Functionality

We will analyze different cases of priority donation/inversion: from simple cases where 2 threads are fighting for the same resources, to cases with multiple locks and nested donation.

An ideal design would be functional in all theses cases, so we will need to take into consideration each particular case and develop a solution for it. The tests will also help us in determining which cases we should focus on.

1. Case 1: Two threads and a lock (Tests 1 and 2)
   This simple case is described at use case 1, section 3.1.3. A solution to this case is to check if the thread with higher priority can access the lock. If it can access it, we won't need priority donation. In contrast, if the LockTryAcquire() function returns false, we will need to implement priority donation:

   - We add the thread with the higher priority to the lock's waiting list
   - We set the priority of the lower priority thread equal to the higher priority thread's priority
   - We wait for the thread to finish it's execution, remove it as the LockHolder and set it's priority back to it's original priority
   - The other thread checks if the owner of the lock is the thread with the lower priority and if false, acquires the lock and removes itself from the waiting list.

   The following pseudocode is possible solution the first case:

```
if $!t2.LockAcquire()$ then
    $WaitingList[] \leftarrow t2$
    if $t2.priority > t1.priority$ and $LockHolder(t1)$ then
        $t1.priority \leftarrow t2.priority$
        while $t1.State = Running$ do
            /* somewhere in this while a LockRelease(t1) will happen
and it's priority will be reverted back to it's initial value */
        end while
        if $LockHolder(NULL)$ then
            t2.LockAcquire()
            $WaitingList[] \rightarrow t2$
        end if
    else
        ... other problem...
    end if
end if
```

In order to pass test 2, we must also implement a function that blocks the thread t1 from lowering it's priority from the moment another thread donated it's priority until it finishes it's execution.

2. Case 2: Multiple threads and multiple locks (Tests 3,4,5 and 6)
This case is described at use case 2, section 3.1.3. It is similar to case 1, except this time we will use succeeding donations to the first thread. We will further describe a solution for the case at test 3:

   - We have thread t1 with the default thread Priority (16) that spawn 2 threads: t2 (with priority 20) and t3(with priority 28). In the initial state, t1 holds both locks with t2 trying to acquire lock1 and t3 trying to acquire lock2. As a consequence lock 1 has the following proprieties: Owner: t1; WaitingList{t2} and lock2: Owner: t1; WaitingList{t3}
   - Since both t2 and t3 are waiting for t1 which is a lower priority thread, a priority donation must occur.
   - In order to implement a correct solution, we must pick the correct thread that donates it's priority. We will pick the thread with the **highest priority**. In our case, t3 will donate it's priority to t1 then after t1 finishes execution , t3 will become the owner of the lock and removes itself form the waiting queue. T1 reverts itself to it's original priority.
   - The same thing will happen with T2: it will donate it's priority to t1 then after it's execution, t2 will become the owner of the lock and removes itself form the waiting queue. T1 reverts itself to it's original priority and finally finishes it's execution.

Functions used:

(a) ThreadInit, ThreadCreate - initialises the threads

(b) ThreadWaitForTermination - so that t3, respectively t2 wait for t1 with donated priority to finish the execution

(c) ThreadGetPriority - to check the thread's priority; will return the donated priority during donation

(d) ThreadSetPriority- used in priority donation: t3 and t2 will donate their priorities to t1.

(e) MutexAcquire - All threads will use this function when requesting the usage of the mutex. When t2 and t3 will use this function, they will be put in the waiting queue.

(f) MutexRelease - Releases the mutex and if a thread is in the waiting list, it becomes the holder

3. Case 3: Nested Donation (tests 7 and 8)
   This case is different from the multimple donations case: instead of multiple threads waiting for a single thread, we have multiple threads that are waiting in chain one after the other. Considering we have three threads: t1,t2,t3 in ascending priority. T3 is waiting for a lock that t2 holds, while t2 is waiting for a lock that t1 holds. In this care it is necessary that t3 donates it's priority to both t2 and t1 such that t1 finishes first, then t2 then t3.

   In all cases when multiple threads are waiting for the same lock, they will be put in order of priority in the waitingList which will then be processed in FIFO manner.

### 3.2.3 Explanation of Your Design Decisions

I decided to focus more on the HAL9000 tests and develop a design focused on them. While i believe a better design would be one that doesn't focus on the particular test cases, and instead is focused on the task as a whole (implementing priority inversion), I believe it is easier to focus on the tests because they cover most of the problems of priority donation.

## 3.3 Tests

1. **TestThreadPriorityDonationOnce**

   - 1. A default priority thread is spawned
   - 2. The default priority thread initializes and acquires a mutex
   - 3. The default priority thread creates a higher priority thread
   - 4. The higher priority thread tries to acquire the mutex taken by the lower priority thread and donates its priority.
   - 5. The default priority thread has now inherited the priority of the newly spawned thread. (THIS IS CHECKED)

     A simple test that checks if a a thread donated it's priority to another thread. It is the first use-case described at section 3.1.3

2. **TestThreadPriorityDonationLower**

   - 1. A default priority thread is spawned
   - 2. The default priority thread initializes and acquires a mutex
   - 3. The default priority thread creates a higher priority thread
   - 4. The higher priority thread tries to acquire the mutex taken by the lower priority thread and donates its priority.
   - 5. The default priority thread has now inherited the priority of the newly spawned thread.
   - 6. The initial thread tries to lower its priority to ThreadPriorityLowest. Its priority should still remain the one of the donators thread. (THIS IS CHECKED)

3. **TestThreadPriorityDonationMulti**
   The main thread (ThreadPriorityDefault) creates and acquires 2 mutexes and spawns 2 additional threads with higher priorities: ThreadPriorityDefault + 4 and ThreadPriorityDefault + 8. The first spawned thread tries to acquire the first mutex, while the second one the second mutex. As a result, after each thread is spawned the main thread receives a priority donation. The main thread then releases the second mutex and validates if its priority is that of the first thread spawned after which it releases the first mutex and validates its priority to be the one with which it started execution.

   This is a test for the case of multiple mutexes.

4. **TestThreadPriorityDonationMultiInverted**
   Same as "TestThreadPriorityDonationMulti", except the mutexes are released in a different order: the second mutex acquired is also the second one released and the first mutex acquired is the first one released.

5. **TestThreadPriorityDonationMultiThreads**
   Same as "TestThreadPriorityDonationMulti" except there are now 3 threads waiting on two mutexes. The two higher priority ones are waiting for the second mutex, while the first spawned thread is waiting on the first mutex.

6. **TestThreadPriorityDonationMultiThreadsInverted**
   Same as "TestThreadPriorityDonationMultiThreads", except the mutexes are released in a different order (as in the "TestThreadPriorityDonationMultiInverted" test)

7. **TestThreadPriorityDonationNest**
   The main thread with ThreadPriorityDefault initializes 4 mutexes and acquires the first one (i.e. Mutex[0]). It then spawns 3 additional threads T[i]: each with priority ThreadPriorityDefault + i (where i is from 1 to 3). Each thread will acquire Mutex[i] and Mutex[i+1
   T[1] takes Mutex[1] and then tries to take Mutex[0]

T[2] takes Mutex[2] and then tries to take Mutex[1]
T[3] takes Mutex[3] and then tries to take Mutex[2]

As a result of these operations the main thread will receive T[1]'s priority because it holds Mutex[0]. T[1] also receives T[2]'s priority because it holds Mutex[1] and as a result T[2]'s priority will be donated to the main thread.
The same happens for T[2]...

8. **TestThreadPriorityDonationChain**
   Same scenario as in "TestThreadPriorityDonationNest", except the number of additional threads is 7.

## 3.4 Observations

# Capitolul 4

# Design of Module *virtualmemory*

## 4.1 Assignment Requirements

Implement the system calls SyscallVirtualAlloc(...) and SyscallVirtualFree(...),
whose signatures are given in file "syscall$_f$*unc.h*".*Youshouldonlyhandlecaseswhenthefollowingco*
*BaseAddress == NULL(i.e.letthekerneldecidewhereinthecallingprocess'virtualaddressspacet*
*NULL(i.e.notamemory − mappedfile), andKey == 0(nosharing).NOTE :*
*youcouldmakeusethekernelfunctions"VmmAllocRegionEx(...)"and"VmmFreeRegionEx(...)"*
*PageFaultNo);", whichstoresin"PageFaultNo"thenumberofpagefaultsgeneratedduringacces*
*AllocatedPhysAddr);", whichstoresin"AllocatedPhysAddr"thephysicaladdressthegiven"Allocc*
*IntFragSize);", whichstoresin"IntFragSize"thenumberofbyteslostduetointernalfragmentati*

### 4.1.1 Initial Functionality

### 4.1.2 Requirements

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

- *Per Process Quotas.* You have to limit the number of open files a process can have and the number of physical frames it may use at one time.

- *System calls.* You have to implement two new system calls for allocating and de-allocating virtual memory. You must also support shared memory using this implementation.

- *Swapping.* You have to implement a page replacement algorithm (i.e. second chance) and swapping.
- *Zero Pages.* You have to implement support for allocating virtual pages filled with zero bytes.

The way to allocate requirements on member teams.

- 3-members teams
    1. Per Process Quotas + Zero Pages
    2. System calls
    3. Swapping

### 4.1.3 Basic Use Cases

## 4.2 Design Description

### 4.2.1 Needed Data Structures and Functions

### 4.2.2 Detailed Functionality

Some questions you have to answer (taken from the original Pintos design templates):

1. Process Quotas
    - When sharing memory between processes the physical frames used should be counted in the quota for each process, how will you achieve this?
2. page table management
    - In a few paragraphs, describe your code for locating the frame, if any, that contains the data of a given page.

- How does your code coordinate accessed and dirty bits between kernel and user virtual addresses that alias a single frame, or alternatively how do you avoid the issue?
- When two user processes both need a new frame at the same time, how are races avoided?

3. page replacement and swapping

   - When a frame is required but none is free, some frame must be evicted. Describe your code for choosing a frame to evict.
   - When a process P obtains a frame that was previously used by a process Q, how do you adjust the page table (and any other data structures) to reflect the frame Q no longer has?
   - Explain the basics of your VM synchronization design. In particular, explain how it prevents deadlock. (Refer to the textbook for an explanation of the necessary conditions for deadlock.)
   - A page fault in process P can cause another process Q's frame to be evicted. How do you ensure that Q cannot access or modify the page during the eviction process? How do you avoid a race between P evicting Q's frame and Q faulting the page back in?
   - Suppose a page fault in process P causes a page to be read from the file system or swap. How do you ensure that a second process Q cannot interfere by e.g. attempting to evict the frame while it is still being read in?
   - Explain how you handle access to paged-out pages that occur during system calls. Do you use page faults to bring in pages (as in user programs), or do you have a mechanism for "locking" frames into physical memory, or do you use some other design? How do you gracefully handle attempted accesses to invalid virtual addresses?
   - A single lock for the whole VM system would make synchronization easy, but limit parallelism. On the other hand, using many

locks complicates synchronization and raises the possibility for deadlock but allows for high parallelism. Explain where your design falls along this continuum and why you chose to design it this way.

4. Shared Memory

   - What data do you need to maintain in order to be able to create and access shared memory regions? What data structure will you be using to maintain the keys?
   - After creating a shared memory region, when it is accessed by the second process does the same virtual address need to be returned to the process? Motivate your answer.
   - How will you know when to delete the shared memory region? Take this scenario as an example: a process creates a region and 7 other processes access it, i.e. they call *SyscallVirtualAlloc* with the same key, how do you know when you need to delete the key from your list and implicitly the shared memory region? This region should be valid until the last process which 'opened' it 'closes' it, i.e. calls *SyscallVirtualFree*.

5. Zero Pages

   - How will you implement zero pages? How will you support accessing a range of zero memory larger than the swap file? Explain the algorithm required for your implementation.

#### 4.2.3 Explanation of Your Design Decisions

## 4.3 Tests

## 4.4 Observations

# Capitolul 5

# Design of Module *virtualmemory*

## 5.1 Assignment Requirements

1. Implement the system calls SyscallVirtualAlloc(...) and SyscallVirtualFree(...), whose signatures are given in file "syscall_func.h".

2. Add a new system call "STATUS SyscallGetPageFaultNo(IN PVOID AllocatedVirtAddr, OUT QWORD* PageFaultNo);", which stores in "PageFaultNo" the number of page faults generated during accesses to the virtual page containing the address given by the "AllocatedVirtAddr" parameter.

3. Add a new system call "STATUS SyscallGetPagePhysAddr(IN PVOID AllocatedVirtAddr, OUT PVOID* AllocatedPhysAddr);", which stores in "AllocatedPhysAddr" the physical address the given "AllocatedVirtAddr" is mapped to. If the page containing the given virtual address is not mapped (i.e. not present) in the physical memory, NULL should be returned.

4. Add a new system call "STATUS SyscallGetPageInternalFragmentation(IN PVOID AllocatedVirtAddr, OUT QWORD* IntFragSize);", which stores in "IntFragSize" the number of bytes lost due to internal fragmentation (i.e. space not required, yet allocated) in the virtual

page the given "AllocatedVirtAddr" belongs to. This could occur when the size in bytes of the allocated memory is not a multiple of page size.

5. Test your implementation writing a testing user application.

### 5.1.1 Initial Functionality

We have multiple functions and structures defined in **vmm.h** that will help us in solving our tasks.
**VmmAllocRegionEx:** Allocates a region of virtual memory.
**VmmFreeRegionEx:** Frees a region of memory previously allocated. **MmuSolvePageFault:** Solves a page fault for a given faulty address. **VmmGetPhysicalAddressEx:** Gets the physical address based on a given virtual address

### 5.1.2 Basic Use Cases

Virtual memory has a very important role in the operating system. It allows us to run more applications on the system than we have enough physical memory to support. Virtual memory is simulated memory that can be written to a file

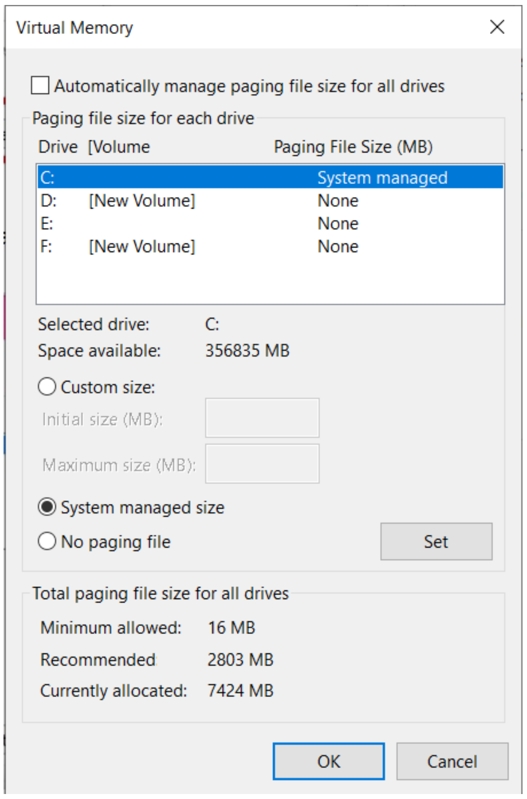
The primary benefits of virtual memory include freeing applications from having to manage a shared memory space, increased security due to memory isolation, and being able to conceptually use more memory than might be physically available, using the technique of paging.

Windows 10, for example, uses this technique of paging. (figure 5.1)

## 5.2 Design Description

### 5.2.1 Needed Data Structures and Functions

Functions: SyscallVirtualAlloc(), SyscallVirtualFree(), SyscallGetPageFaultNo(), SyscallGetPagePhysAddr(), SyscallGetPageInternalFragmentation().
Data structures: PCPU - for counting the page faults



Virtual Memory

- [ ] Automatically manage paging file size for all drives

Paging file size for each drive

| Drive | [Volume | Paging File Size (MB) |
|---|---|---|
| C: | | System managed |
| D: | [New Volume] | None |
| E: | | None |
| F: | [New Volume] | None |

Selected drive: C:
Space available: 356835 MB

○ Custom size:
Initial size (MB):
Maximum size (MB):
● System managed size
○ No paging file

Total paging file size for all drives

| | |
|---|---|
| Minimum allowed: | 16 MB |
| Recommended | 2803 MB |
| Currently allocated: | 7424 MB |

Figura 5.1: Windows 10 virtual memory settings

### 5.2.2 Detailed Functionality

For our implementation, we will take each task, one by one and propose a solution.

1. Implement the system calls SyscallVirtualAlloc(...) and SyscallVirtualFree(...).

   For this functions, we can make use of the VmmAllocRegionEx() and VmmFreeRegionEx() functions. When the SyscallVirtualAlloc() is called, we must initialize the parameters BaseAddress and FileHandle with null and Key with 0, then call SyscallVirtualAlloc(). The same must be done with SyscallVirtualFree().

2. Add a new system call "STATUS SyscallGetPageFaultNo(IN PVOID

AllocatedVirtAddr, OUT QWORD* PageFaultNo);

For this we will use page fault handler in mmusolvepagefault(). An idea would be to count how many times this function is called for a respective faulty memory address. We must pay extra attention to store the faulty address for this counter before each call of the mmusolvepagefault(), because more processes can access the same address.

3. Add a new system call "STATUS SyscallGetPagePhysAddr(IN PVOID AllocatedVirtAddr, OUT PVOID* AllocatedPhysAddr)

   We already have the function VmmGetPhysicalAddressEx() that retrieves the physical address corresponding to VirtualAddress. In our syscall, we will call the said function and check if PHYSICALADDRESS is not null.

4. Add a new system call "STATUS SyscallGetPageInternalFragmentation(IN PVOID AllocatedVirtAddr, OUT QWORD* IntFragSize);
   In order to implement this, we will need SysCallVirutalAlloc to be working. If this call was successfull, then we simply initialize IntFragSize with the remaning unused memory.
   IntFragSize← $AllocatedVirtAdd \, \% \, PAGESIZE$

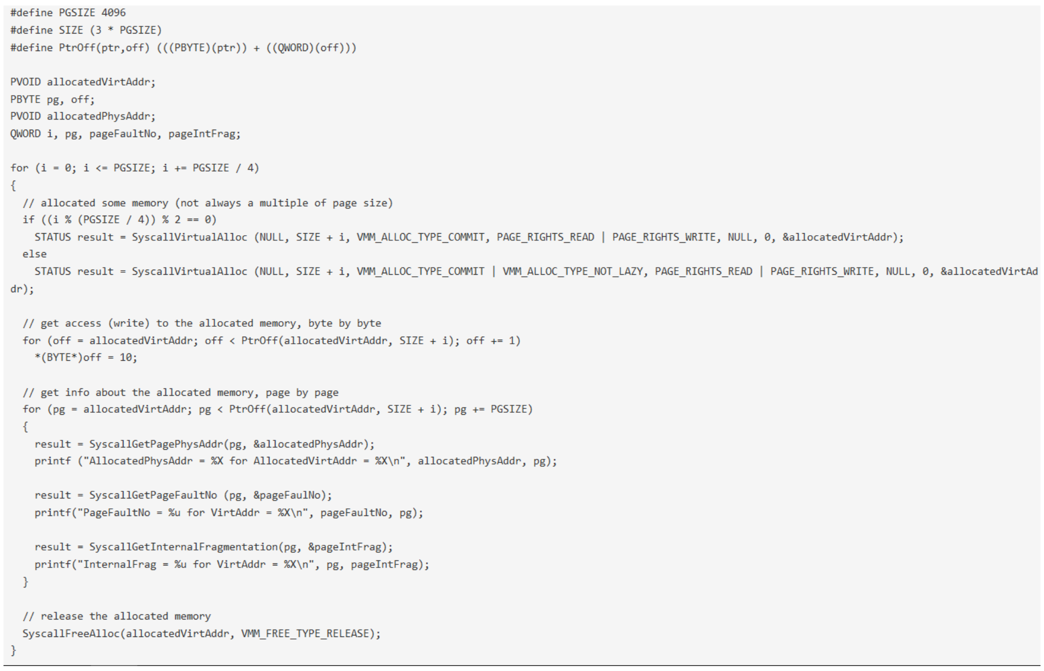
5. Test your implementation writing a testing user application
   This is detailed in section 5.3.

### 5.2.3 Explanation of Your Design Decisions

## 5.3 Tests

To test our functions, we were given the code seen in picture 5.2. Before adding different functions to our test, we need to LOG our progress trough the test. Similarly to the printf functions, we will ad some LOG functions with to scope to track our progress trough a test and identify a possible crash or faulty behaviour.

If we want to test some more functionality, and addition to our tests we can check if a page fault has been resolved after detecting a faulty address. This can be done by calling the SyscallGetPageFaultNo and checking if PageFaultNo is non null. If true, we will call VmmSolvePageFault as long PageFaultNo is not 0 or if the call fails. If at the end the SyscallGetPageFaultNo for the faulty address has PageFaultNo 0, the test passes, else it fails.



```
#define PGSIZE 4096
#define SIZE (3 * PGSIZE)
#define PtrOff(ptr,off) (((PBYTE)(ptr)) + ((QWORD)(off)))

PVOID allocatedVirtAddr;
PBYTE pg, off;
PVOID allocatedPhysAddr;
QWORD i, pg, pageFaultNo, pageIntFrag;

for (i = 0; i <= PGSIZE; i += PGSIZE / 4)
{
  // allocated some memory (not always a multiple of page size)
  if ((i % (PGSIZE / 4)) % 2 == 0)
    STATUS result = SyscallVirtualAlloc (NULL, SIZE + i, VMM_ALLOC_TYPE_COMMIT, PAGE_RIGHTS_READ | PAGE_RIGHTS_WRITE, NULL, 0, &allocatedVirtAddr);
  else
    STATUS result = SyscallVirtualAlloc (NULL, SIZE + i, VMM_ALLOC_TYPE_COMMIT | VMM_ALLOC_TYPE_NOT_LAZY, PAGE_RIGHTS_READ | PAGE_RIGHTS_WRITE, NULL, 0, &allocatedVirtAd
dr);

  // get access (write) to the allocated memory, byte by byte
  for (off = allocatedVirtAddr; off < PtrOff(allocatedVirtAddr, SIZE + i); off += 1)
    *(BYTE*)off = 10;

  // get info about the allocated memory, page by page
  for (pg = allocatedVirtAddr; pg < PtrOff(allocatedVirtAddr, SIZE + i); pg += PGSIZE)
  {
    result = SyscallGetPagePhysAddr(pg, &allocatedPhysAddr);
    printf ("AllocatedPhysAddr = %X for AllocatedVirtAddr = %X\n", allocatedPhysAddr, pg);

    result = SyscallGetPageFaultNo (pg, &pageFaulNo);
    printf("PageFaultNo = %u for VirtAddr = %X\n", pageFaultNo, pg);

    result = SyscallGetInternalFragmentation(pg, &pageIntFrag);
    printf("InternalFrag = %u for VirtAddr = %X\n", pg, pageIntFrag);
  }

  // release the allocated memory
  SyscallFreeAlloc(allocatedVirtAddr, VMM_FREE_TYPE_RELEASE);
}
```

Figura 5.2: Given test

## 5.4 Observations