The Design of *HAL9000* Operating System

Darius-Andrei Moldovan, Tudor Moldovanu, Andrei Suciu

December 1, 2025

Abstract

The abstract should contain a very short description of the results presented in this report.

Take into account that a design document is a high-level, logical (abstract) description of the solution you propose to the problems and requirements you dealt with. Though, it should be detailed enough such that somebody who already knows and understands the requirements could implement (i.e. write the code) the design without having to take any significant additional design decisions. Even further, if such an implementer would happen to be an experienced one, he or she should be able to use the design document to implement the solution in just few hours (not necessarily including the debugging).

Important notes. Take care to remove from the given design document template all the text that was given to you as a guideline and provide your own document with only your own original text. Also, do not simply write anything in any section just to have some text there, but only write text that makes sense in the given context. We will not count the number of pages or words of your document, but will only evaluate the meaning and completeness of the written words.

Chapter 1

General Presentation

1.1 Working Team

Specify the working team members' names and their responsibility.

1. Darius-Andrei Moldovan
   (a) Threads: dealt with Fixed Priority Scheduler
   (b) Userprog: dealt with System Calls for Process Management and File System Access
   (c) VM: dealt with Per Process Quotas and Zero Pages
2. Tudor Moldovanu
   (a) Threads: dealt with Timer
   (b) Userprog: dealt with Argument Parsing and Validation of System Call Arguments
   (c) VM: dealt with Syscalls and Stack Growth
3. Andrei Suciu
   (a) Threads: dealt with Priority Donation
   (b) Userprog: dealt with System Calls for Thread Management
   (c) VM: dealt with Swapping

Chapter 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) where 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 2.1 below to see the way images are inserted in a Latex file.

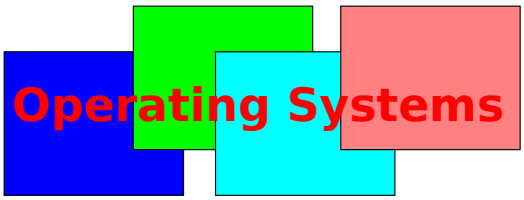

Figure 2.1: Sample image

When describing algorithms you are to use or develop, it is very important to also describe them in a formal way, not just as free text description. Commonly, this could be done as a sort of pseudo-code or just as a simple list of logically ordered steps (enumerated list). In your algorithm description you should (i.e. must) refer to the data structures and variables you described in the previous section, following the recommendations in Section **??**. This way you avoid describing your algorithms too vague and reduce the risks of being misunderstood.

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.

Chapter 3

Design of Module *Threads*

3.1 Assignment Requirement

3.1.1 Initial Functionality

Timer

We are given an implementation of the executive timer (see file `ex_timer.c`), which is based on busy-waiting, i.e. a loop where the time expiration condition is checked continuously, keeping the CPU busy.

Priority Scheduler. Fixed Priority Scheduler

We are given an implementation of a Round-Robin scheduling policy (see function `_ThreadGetReadyThread()` in file `thread.c`), which consider all threads equal, giving them CPUs based on FCFS (First-Come First-Served) principle and only for a predefined time slice. We are also given the implementation of thread switching mechanism (see function `ThreadSwitch` in file `_thread.yasm`).

Priority Scheduler. Priority Donation

Same given code as described at previous section.

3.1.2 Requirements

Timer

We must change the given executive timer functionality, such that to not use anymore the busy waiting technique in function `ExTimerWait()`, but a

blocking mechanism, which suspends the waiting thread (i.e. takes the CPU from it) until the waited timer expires.

Priority Scheduler. Fixed Priority Scheduler

We must implement a priority-based thread scheduling policy (algorithm), which means the scheduler must consider threads' priorities when deciding which thread to be given an available CPU. The main scheduling rule is that when a thread must be chosen, the one with the higher priority must be the choice. The scheduler must be preemptive, which means it must always assure that threads with the highest priorities (considering multiple CPUs) are the ones run at any moment. This could suppose a currently running thread could be suspended, when a new higher-priority thread occurs.

Priority Scheduler. Priority Donation

We must implement (temporary) priority donation as a solution to the "priority inversion" problem. Priority inversion correspond to situations when a thread with a higher priority wait for a thread with a smaller priority (contrary to the priority-based rule, which requires the opposite). Such a situation occur when a thread with a smaller priority succeeds taking a lock, which will be later on required by a higher-priority thread. In order to avoid having the higher-priority thread waiting for smaller-priority threads (others than the lock holder, but having a higher priority than it), the higher thread donates its priority to the lock holder (smaller thread) until the lock will be released. In the meantime, no other "in-between" thread, could block the lock holder and consequently the waiting higher-priority thread.

3.1.3 Basic Use Cases

Timer

A user application could use a timer to have one of its threads periodically (e.g. every one second) increasing a counter and displaying it on the screen. This could be a sort of wall-clock.

Priority Scheduler. Fixed Priority Scheduler

A user application could establish different priorities for its different threads, based on some application specific criteria. Similarly, the OS itself could create a (kernel) thread to handle critical system events, giving that thread a higher priority than those of all other user-application threads.

Priority Scheduler. Priority Donation

This is not directly visible to and controlled by a user application, but has effects on scheduling performance, in the sense that high-priority threads will not be delayed too much when competing for locks with smaller-priority threads (already having those locks). It could be difficult to create a use case (in a user application) to measure the correctness and effectiveness of such a mechanism, but we could image a use case in kernel (as we already have in the given tests). An example in this sense could be the following scenario:

1. start a testing thread, which will take the following steps
2. creates a smaller-priority thread, which takes a lock;
3. after being sure the lock was taken by the smaller-priority thread, creates a second, higher priority thread, which wants to take the same lock; we must see the current lock holder is given (donated) the higher-priority thread;
4. while the smaller-priority thread is still keeping the lock, other threads, with priorities between the small and high threads could be created, which we must see that will not suspend the small-priority thread holding the lock;
5. the small-priority thread releases the lock, and we must see that its priority goes back to its original one;
6. the main thread waits until all created threads terminates, before terminating itself.

3.2 Design Description

3.2.1 Needed Data Structures and Functions

Timer

We will change the `_EX_TIMER` structure in the following way:

```
struct _EX_TIMER
{
    ...

    // keep track of threads waiting (blocked) for the timer
    EX_EVENT    TimerEvent;
```

```
    // used to place the timer in a global timer list
    LIST_ENTRY  TimerListElem;

    ...
} EX_TIMER, *PEX_TIMER;
```

We add a global list keeping track of all timers in the system and a lock to protect this list while accessed concurrently:

```
struct _GLOBAL_TIMER_LIST
{
    // protect the global timer list
    LOCK                TimerListLock;

    // the list's head
    LIST_ENTRY          TimerListHead;

};

static struct _GLOBAL_TIMER_LIST m_globalTimerList;
```

Functions that we will change:

- `ExTimerInit()`: add the new timer in the global timer list;
- `ExTimerStop()`: signal waiting threads, the timer is no longer evolving;
- `ExTimerWait()`: replace the busy-waiting with the blocking technique;
- `ExTimerUninit()`: remove the timer from the global timer list.

New functions that we will add to file `ex_timer.c`:

- `ExTimerSystemPreinit(void)`: initialize the global timer list and its associated lock;
- `ExTimerCompareListElems(PLIST_ENTRY t1, PLIST_ENTRY t2, PVOID context)`: compare two timer trigger time in order to keep the global timer list order by timer triggering time;
- `ExTimerCheck(PEX_TIMER timer)`: called on each timer interrupt handling to check for a particular timer if it must be triggered or not;
- `ExTimerCheckAll(void)`: called on each timer interrupt handling to check for all timers in the global timer list if they must be triggered or not.

Priority Scheduler. Fixed Priority Scheduler

We need to take into account the threads' priorities, but as long as this is a field already existing in the `_THREAD` structure, we must only use it.

```
typedef struct _THREAD
    ...

    THREAD_PRIORITY          Priority;

    ...
};
```

In order to keep track of the minimum priority of all threads running at one moment, we will keep a global variable `RunningThreadsMinPriority` for this, placed in the `_THREAD_SYSTEM_DATA` structure. Because we also work on the `ReadyThreadsList`, we also illustrate it below:

```
typedef struct _THREAD_SYSTEM_DATA
    ...
    _Guarded_by_(ReadyThreadsLock)
    LIST_ENTRY           ReadyThreadsList;

        _Guarded_by_(ReadyThreadsLock)
    THREAD_PRIORITY      RunningThreadsMinPriority;
};
```

The functions (all in `thread.c`) we will make changes in or just use are:

- `ThreadSystemPreinit()`;
- `ThreadYield()`: change it such that to consider thread priorities;
- `ThreadUnblock()`: change it such that to consider thread priorities; in particular when a thread with a higher priority of any other running thread is unblock, one of the smaller-priority running threads must be preempted;
- `MutexAcquire()`: keep the mutex's waiting queue ordered by thread priorities, such that when the mutex become available, the thread with the highest priority in the mutex's waiting queue to be unblocked;
- `ExEventWaitForSignal()`: to keep the event's waiting list ordered by thread priorities.
- `ThreadSetPriority()`: force the currently running thread give up the CPU, if its new priority is smaller than that of any thread in the ready list;

- `SmpSendGenericIpi()`: send an inter-processor interrupt to require all CPUs recheck the ready list, such that to be sure that the running threads are always the ones with the highest priority; used when a new thread is unblocked.

The new functions that we will add are:

- `ThreadComparePriorityReadyList(PLIST_ENTRY e1, PLIST_ENTRY e2, PVOID Context)`: compare the two threads, `t1` and `t2`, based on their priority; will be used to keep the thread lists ordered based on their priorities (descending order);
- `ThreadYieldForIpi`: to be run be CPUs, when receiving an IPI (inter-processor interrupt), when a running thread must be preempted by a unblocked thread.

Priority Scheduler. Priority Donation

Priority donation supposes giving a thread a new temporal priority, while that thread is holding a mutex. We must be able to restore such a thread's priority back to its original one, so logically, we must to keep track of both types of priorities. The two priorities are called in related literature (or other real OSes) the *real priority* (or original priority), the priority established at thread creation or changed during thread execution by the thread itself, and the *effective priority* (or actual priority), the priority dynamically established by the OS (based on different internal criteria) and considered by the priority-based OS' scheduler. Based on this basic deduction and on the analysis described in Section 3.2.3, we established the need for the following new fields in existing data structure or new data structures. While there were already a field "`Priority`" associated to each thread, and some of the tests consider it as the effective one, we let its name unchanged, and only added a new field for what we consider to be the real priority.

In file `thread_internal.h`:

```
struct _THREAD
{
    LOCK                PriorityProtectionLock;

    _Guarded_by_(PriorityProtectionLock)
    THREAD_PRIORITY     Priority;       // the effective priority (alread

    ...
```

```
    // Used for priority donation
    _Guarded_by_(PriorityProtectionLock)
    THREAD_PRIORITY     RealPriority;     // the real (original) priority

    LIST_ENTRY          AcquiredMutexesList; // the list of mutexes held
    PMUTEX              WaitedMutex;      // the mutex thread waits for

    ...
}
```

In file `mutex.h`:

```
struct _MUTEX
    ...
    LIST_ENTRY      AcquiredMutexListElem; // elem in list of mutexes acq
    ...
} MUTEX, *PMUTEX;
```

Functions that must be changed (in `mutex.c` and `thread.c`) are:

- `MutexAcquire()`: donate priority of blocking thread (in case the mutex in already acquired), if its priority is higher than that of mutex holder;
- `MutexRelease()`: recompute the priority of the thread releasing the mutex;
- `ThreadSetPriority()`: take into account both real priority, which is changed by the functions, and the current effective (donated) one;
- `ThreadGetPriority()`: assure the actual (i.e. effective, donated, if the case) priority is the one returned;

New functions that we will implement (in `thread.c`) are:

- `ThreadDonatePriority()`: called in `MutexAcquire()` to donate priority to a mutex holder and also deal with the nested donation aspect;
- `ThreadRecomputePriority()`: called in `MutexRelease()` to recompute the priority of a thread just releasing a mutex, taking into account its real priority, but also priorities donated due to other mutexes that thread still holds.

3.2.2 Interfaces Between Components

Timer <-> Fixed Priority Scheduler:

- `ExTimerInit()`: No calls between components.
- `ExTimerStop()`: No calls between components.
- `ExTimerWait()` <-> `ThreadBlock()`: We don't use anymore the busy-waiting technique; we will call `ThreadBlock()` to suspend the thread and remove it from the active execution list, allowing other threads to execute in the meantime.
- `ExTimerUninit()`: No calls between components.
- `ExTimerSystemPreinit(void)`: No calls between components.
- `ExTimerCompareListElems(PLIST_ENTRY t1, PLIST_ENTRY t2 , PVOID context)`: No calls between components.
- `ExTimerCheck(PEX_TIMER timer)`: No calls between components.
- `ExTimerCheckAll(void)` <-> `ThreadUnblock()`: Examines all active timers; we will call `ThreadUnblock()` to unblock the thread and add it back to the active execution list, where the priority scheduler will handle its execution based on priority.

Timer <-> Priority Donation

- `ExTimerInit()`: No calls between components.
- `ExTimerStop()` <-> `ThreadSetPriority()`: When a thread is blocked by a timer which expires, if its Real and Effective priorities are different, its priority will be reset to the Real priority.
- `ExTimerWait()`: No calls between components.
- `ExTimerUninit()`: No calls between components.
- `ExTimerSystemPreinit()`: No calls between components.
- `ExTimerCompareListElems()`: No calls between components.
- `ExTimerCheck()`: No calls between components.
- `ExTimerCheckAll(void)`: No calls between components.

Fixed Priority Scheduler <-> Priority Donation

- `ThreadSystemPreinit()`: No calls between components;
- `ThreadYield()` <-> `ThreadDonatePriority()`: When a thread wishes to yield to the CPU, the function will check if it has any mutexes and if any threads are waiting for it, in which case it will donate the priority to those waiting.
- `ThreadUnblock()`: <-> `ThreadSetPriority()`: When a thread is unblocked, if its Real and Effective priorities are different, its priority will be reset to the Real priority.
- `MutexAcquire()`: No calls between components.
- `ExEventWaitForSignal()`: No calls between components.
- `ThreadSetPriority()`: No calls between components.
- `SmpSendGenericIpi()`: No calls between components.
- `ThreadComparePriorityReadyList()`: No calls between components.
- `ThreadYieldForIpi()`: No calls between components.

3.2.3 Analysis and Detailed Functionality

Timer

Replacing the busy-waiting requires a mechanism to block a thread until the waited event (i.e. a particular time moment) occurs. We immediately noted the function `ThreadBlock()` (in file `thread.c`) provides such a functionality. We took a look of places `ThreadBlock()` was already used, to see the way the blocking mechanism is used. One good example was in function `MutexAcquire()`, where we noted that before blocking the current thread (i.e. the one waiting for the mutex), it was inserted in a waiting list, associated to that mutex. We also noted that the blocked thread was unblocked in function `MutexRelease()`, by calling the function `ThreadUnblock()` on the blocked thread removed from the waiting list. So, our first idea was to create a similar list for each timer instance, where to keep threads waiting for that timer to expire, using `ThreadBlock()` function to block a thread and `ThreadUnblock()` to unblock it.

However, we noted that the `ThreadBlock()` function was called in function `ExEventWaitForSignal()` also. Investigating a little further, we noted

that the executive event was a general wait to manage threads waiting for a particular (not specified, so general) event, by blocking the waiting threads and unblocking them when the event occurred (i.e. signaled). This seemed to be a higher level interface to the block / unblock mechanism, so we decided to use it.

As a result, we decided to associate an executive event to each timer. This translates in an additional `EX_EVENT` field in the `_EX_TIMER` structure, like illustrated in Section 3.2.1.

Once we had established the `TimerEvent` field, we had to determine three aspects related its usage:

1. when and how to initialize it;
2. where and how to use it to block a thread waiting for the timer to expire;
3. where and how to use it to unblock the waiting threads, when the timer expires.

The decisions we took in these sense were the following ones:

1. logically, we could (and must) initialize a timer's event, when that timer is initialized itself, i.e. in the function `ExTimerInit()`; we could do it by calling the `ExEventInit()` function, with the event type "`ExEventTypeNotification`" and not signaled initially.
2. replacing the busy-waited loop and blocking the waiting thread could be done very simple in function `ExTimerWait()`, by simply calling the function `ExEventWaitForSignal()` on the timer's event;
3. unblocking the thread should be done very simple by calling the `ExEventSignal()` function of the timer's event, though where to do this was not pretty obvious; in order to determine this, we looked in more details at the condition checked by the initial code in the busy waiting loop in function `ExTimerWait()` and noted there were actually two conditions:
   (a) one (the second, actually) that checked in the timer was still started (checking the `Timer->TimerStarted` field); this immediately lead us to the conclusion that one place to call `ExEventSignal()` was the function `ExTimerStop()`, where the `TimerStarted` field was set to FALSE;
   (b) another one that checked if the system time (returned by function `IomuGetSystemTimeUs()`) had reached a value equal to or bigger

than the timer's triggering time; this suggested us that the other place the ExEventSignal() should be called was where the time passage could be observed; investigating in this direction, we found out that this was related to the timer interrupt occurrence and also that the function ExSystemTimerTick() (in file ex_system.c) was called every time a timer interrupt occurred; so, we decided to have a function ExTimerCheck() to be called from ExSystemTimerTick(); the way that function works is described below.

The function ExTimerCheck() must check if the system's time is bigger than or equal to a timer's triggering time and if the case, call the ExEventSignal() on that timer's event. This looks like this:

```
if (system_time >= timer_triggering_time)
    call ExEventSignal() on timer event field
```

We also noted that the same checking must be done on all possible timers existing in the system, so we needed a way to keep track of all of them and to call their ExTimerCheck() function. This is why we decided to keep a global list with all timers in the system. We also decided to have a lock protecting that list for concurrent accesses (it would be possible for multiple threads running on different CPUs to concurrently create or remove different timers). We placed the list and its lock in a structure called struct _GLOBAL_TIMER_LIST and created a global variable of that type, called m_globalTimerList.

As with any other global variable or data structure field we had to determine where to initialize it and its lock, where to use it (i.e. in our case add new timers or remove them) and where to destroy it. We took the following decisions in this context:

1. create a new function ExTimerSystemPreinit() to be called where other functions initializing other OS's components were called, i.e. in function SystemPreinit() (file system.c); that function would simply initialize the global list (by calling the InitializeListHead() function) and the lock (by calling the LockInit function);

2. adding a new timer in the list will be done in function ExTimerInit, by performing the following steps:

   (a) acquire the lock that protects the global timer list (calling LockAcquire());

   (b) insert the timer's structure in the global timer list (by calling InsertOrderedList(), for reason explained below);

   (c) release the global timer list's lock (calling LockRelease()).

3. removing a timer from the global list will be done in function `ExTimerUninit`, by performing the following steps:
   (a) acquire the lock that protects the global timer list (calling `LockAcquire()`);
   (b) remove the timer's structure in the global timer list (calling `RemoveEntryList()`);
   (c) release the global timer list's lock (calling `LockRelease()`).
4. iterate the global list when a timer interrupt occurs (i.e. in function `ExSystemTimerTick()`) and call the `ExTimerCheck()` for each thread in the global list; actually, in order to keep the current coding style, we will declare the global timer list as static (so visible only in file `ex_timer.c`) and a new function `ExTimerCheckAll()` (in file `ex_timer.c`) to be called from `ExSystemTimerTick()` and performed the mentioned steps.
   (a) acquire the lock that protects the global timer list (calling `LockAcquire()`);
   (b) iterate the global timer list (e.g. by calling `ForEachElementExecute()` or any other way the iterate a list) and call the `ExTimerCheckAll()` for each element in that list;
   (c) release the global timer list's lock (calling `LockRelease()`).

We discover an optimization we could apply when checking for timers' triggering time, in order to shorted the time spent handling the timer interrupt. We noted that only timers having their trigger time smaller than or equal to the system time must be signaled, so we decided to keep the global timer list ordered (ascending) by timers' trigger time. This way, once a timer with its trigger time bigger than the system's time is encountered, the list iteration could be stopped. This is why we will call `InsertOrderedList()` in `ExTimerInit`. That function expects as a parameter a function that know to compare two timers, which we decide to be a new function `ExTimerCompareListElems()`. This function works as follows:

1. obtains from its first two parameters, which are of the generic type `PLIST_ENTRY`, two pointers to `EX_TIMER` structure, using the `CONTAINING_RECORD` macro;
2. compare the two timers by calling the function `ExTimerCompareTimers()`.

Take care that using the `ForEachElementExecute()` function to iterate the global timer list could traverse the entire list, even if that list is ordered, so making your optiomization ineffective. Though, we could change this behaviour setting correspondigly the forth parameter (`AlMustSucceed`) of `ForEachElementExecute()` function and by carrefully implementing the

ExTimerCompareTimer() function. Take a look at the implementation of function ForEachElementExecute() to see exactly how its forth parameter must be specify and how the ExTimerCompareTimer() function must be implemented.

Priority Scheduler. Fixed Priority Scheduler

The priority-based scheduler's policy requires that at any moment the highest priority threads (from the ones wanting for the CPU, i.e. the so-called ready threads) to be running. This rule implies two requirements we must to deal:

1. when there are more threads we must choose from, the one with the highest priority must be chosen;
2. an unblocked or a newly created thread with a higher priority than any other running thread must preempt a smaller priority thread and be given the released CPU.

In order to satisfy the first requirement, we decided to order all thread lists in the system, based on their priorities, in the descendant order, such that to always have the thread with the highest priority in front of the list. We had identified all such lists, which are:

1. the ready list (see field ReadyThreadsList of structure _THREAD_SYSTEM_DATA, file thread.c);
2. mutexes' lists (see field WaitingList of structure _MUTEX, file mutex.h);
3. events' lists (see field WaitingList of structure _EX_EVENT, file ex_event.h).

For keeping the thread lists ordered, we will replace the usage of InsertTailList() with the function InsertOrderedList(), which will be given as parameter a thread comparison function ThreadComparePriorityReadyList() (described below), in the following functions:

- in ThreadUnblock() for inserting the unblocked thread in ReadyThreadsList ordered descending by priority;
- in ThreadYield() for inserting the yielding thread (i.e. the one giving up the CPU) in ReadyThreadsList ordered descending by priority;
- in MutexAcquire() for inserting the blocking thread (i.e. the one waiting for the mutex) in WaitingList ordered descending by priority;
- in ExEventWaitForSignal() for inserting the blocking thread (i.e. the one waiting for the event occurrence) in WaitingList ordered descending by priority.

The `ThreadComparePriorityReadyList()` function works the following way:

- obtains from its first parameter `e1`, which is of the generic type `PLIST_ENTRY`, a pointer to a `_THREAD` structure, using the `CONTAINING_RECORD` macro, whose the first parameter is the list element, the second is the `THREAD` structure containing that list element, and the third is the name of the list element field in the THREAD structure, which is the "`ReadyList`":

```
PTHREAD pTh1;
pTh1 = CONTAINING_RECORD(e1, THREAD, ReadyList);
```

- similarly, obtains from its second parameter `e2` a pointer to a `_THREAD` structure:

```
PTHREAD pTh2;
pTh1 = CONTAINING_RECORD(e2, THREAD, ReadyList);
```

- compare the two threads' priorities and return the result such that to order the list in a descendant way (i.e. negative, if second thread's priority is less the that of the first, positive if the opposite, and zero if equal):

```
prio2 = ThreadGetPriority(pTh2);
prio1 = ThreadGetPriority(pTh1);

compare_and_return_result(prio1, prio2);
```

Having the ready list list ordered by thread priority implied no need to change the `_ThreadGetReadyThread()` function, which chooses a thread from ready list to be given an available CPU, while choosing the first thread in ready list (by calling `RemoveHeadList()`) would return the thread with the highest priority, exactly what the priority-based policy requires.

Another aspect we should take care about when scheduling threads based on their priorities (besides the rule of always choosing the thread with the highest priority from a thread list) is the case more threads have the same priority, in particular, the same highest priority. In such a case, a fair scheduler would give equal chances to all such threads. This could be provided by using the Round-Robin (RR) policy (the one already being implemented in *HAL9000*), which would take the threads in the order they were added to the ready list and will give them CPUs for an establish amount of time (time quantum or slice), placing them back in ready list when their allocated time slice expires. The RR strategy must manage the ready list in a FIFO manner, i.e. appending at the end of the list a thread suspended due to its time

quantum expiration. This could be managed on our priority-ordered ready list, if the `InsertOrderedList()` function would insert a thread with a particular priority after all threads with the same priority already in ready list (i.e. at the end of its priority class), while doing so it would be the equivalent of appending to a list containing only threads of that priority. We looked at the implementation of `InsertOrderedList()` and noted that it complied this requirements based on the the given comparison function (see above), which means that our scheduler would handle threads with the same priority based on the RR policy.

In order to satisfy the preemption requirement, we searched for situations (and corresponding functions) when a thread becomes a new competitor for CPUs, besides the existing one. The threads competing for CPUs are the running threads and the threads in the ready list, having the running threads with a higher priority than all those in ready list. However, a newly arrived thread could have a higher priority than some of those running at that moment and this is why our scheduler should be called to preempt one of the running threads if having a smaller priority than the newly arrived one, or inserting the new thread in the ready list, in the opposite case.

We identifies such situation for:

1. a newly created thread (see function `ThreadCreate()`, which further calls `ThreadCreateEx()`);

2. an unblocked thread (see function `ThreadUnblock()`).

However, if when we looked in more details at function `ThreadCreateEx()` we noted that a newly create thread is set active (i.e. ready) by calling the `ThreadUnblock()` function, which simply inserts that thread in the ready list, similarly to any other existing thread that is unblocked due to the fulfillment of some condition it was waiting for (e.g. a mutex to be released, an event to be signaled — see functions `MutexRelease()` and `ExEventSignal()`, where `ThreadUnblock()` is called). So, we concluded that the `ThreadUnblock()` function is the only place we should impose the preemption functionality of our priority-based scheduler, based on the following additional logic:

Require: `unblocked_thread`, `ready_list`, `min_priority_running_threads`
Ensure: Preempt a running thread if smaller than the unblocked one
  new_prio = ThreadGetPriority(unblocked_thread)
  min_prio = GetMinPriorityOfRunningThreads
  if *new_prio* > *min_prio* **then**
    Preemption() ▷ Preempt one smaller-priority running thread
  else
    InsertOrderedList(ready_list, unblocked_thread)

end if

A CPU preemption could be implemented by using the `SmpSendGenericIpi()` function, sending an interrupt (i.e. IPI) to all CPUs, forcing them interrupt their current execution and compare their currently running thread with those in the ready list.

Sending an IPI to all CPUs could be done the following way:

```
SMP_DESTINATION dest = { 0 };

SmpSendGenericIpiEx(ThreadYieldForIpi, NULL, NULL, NULL,
     FALSE, SmpIpiSendToAllIncludingSelf, dest);
```

The given `SmpIpiSendToAllIncludingSelf()` function would be executed by each CPU when handling the received IPi and should checked the condition mentioned before. We could immediately noted that the function `ThreadYield()` does exactly this, besides protecting the ready list with a lock while investigating it, a condition which is really necessarily to avoid multiple CPUs taking the same threads for being run. Pay attention however, that calling `ThreadYield()` in an interrupt handler (what the `ThreadYieldForIpi()` function basically is for the IPI we send to CPUs) is not a good idea, while switching CPU to another thread from the one currently handling an interrupt could lead to unpredictable results. Yet, you could trigger an `ThreadYield()` indirectly, when the interrupt handling is over, by setting to TRUE the `YieldOnInterruptReturn` flag for the current CPU. See a similar mechanism in `ThreadTick()` function, where the Round-Robin scheduler forces a CPU switch (i.e. a thread yield) when the current thread quantum expires: `pCpu->ThreadData.YieldOnInterruptReturn = TRUE;`.

One aspect we should manage is the way to find out is preemption is needed or not when a thread is unblocked, even in case of concurrent execution of CPUs. This means that while comparing the unblocked thread's priority to those of the currently running threads, we must be sure the investigated CPUs will not be switched to other threads. We found out two ways to manage correctly this concurrency:

1. always send an IPI to all CPUs, letting them call `ThreadYield()`, which is synchronized (as mentioned before); this is the simplest way, though not the most efficient one, as all CPUs will be interrupted, even if the unblocked thread is smaller than all currently running ones, so must not preempt any one of them;

2. keep a protected list of all running threads or just a global variable having the value of the minimum priority of all running threads; the access to that variable must be protected by a lock, which should be

acquired both in `ThreadUnblock()` while making the needed comparison, and when a new thread gets a CPU (a candidate function for this could be `ThreadCleanupPostSchedule()`).

We chose the first alternative, being more simple, which, of course, changed a little bit the preemption logic in `ThreadUnblock()` described above.

After further investigations, we found out that even if our priority-based scheduler does not change the priority of threads, letting them as established at thread creation (supposed to be controlled by user applications those threads belong to), there is still another place (i.e. function), where a thread priority could be changed by that thread itself. This is function `ThreadSetPriority()`, which, if you look at can note could be called only by a running thread to change its own priority. Priority change, immediately make us think of calling the scheduler to reevaluate the situation regarding the threads competing for CPUs. We identified two situations:

1. if a currently running thread calling `ThreadSetPriority()` would increase its priority, this would have changed nothing at all the current situation, while it is supposes that the thread's previous priority had already been higher than those of all threads in ready list (due to the priority-base policy), so increasing it even more, produces no effects;

2. if a currently running thread calling `ThreadSetPriority()` would decrease its priority, there could be two subcases:

   (a) if the new priority is larger than those of all threads in ready list, noting should happen, while this is equivalent to the previous case;

   (b) if the new priority is smaller than one of threads in ready list, then the currently running thread must give up the CPU in favor of a higher-priority thread in ready list; this could done be very simple by calling the `ThreadYield()` function.

Priority Scheduler. Priority Donation

As mentioned in Section 3.2.1, we firstly established the need for two types of priorities associated to a thread: real and effective.

Ignoring the priority donation problem, the real priority is the one established at thread creation (in function `ThreadCreateEx()`) or when the thread itself changes its priority (in function `ThreadSetPriority()`). Correspondingly, we assign values to the real priority field `RealPriority` in function `_ThreadInit()` (called by `ThreadCreateEx`) and in function `ThreadSetPriority()`.

Those values correspond to the functions' priority parameter and, if no priority donation takes place for a thread, are equal to the effective priority field `Priority`.

Though, while we must solve the *priority inversion* problem based on priority donation, we could have at one moment different values for the two priorities and must handle differently them. The priority the scheduler must consider when taking its decisions must be the effective priority, i.e. the donated priority, if a priority is donated to it (logically, does not make sense to donate a higher priority to a thread, if that priority is not considered by scheduler).

The priority inversion problem is usually defined (see the lecture slides related to priority-based scheduling) in relation to mutexes. The main idea is that while holding a mutex, a lower-priority thread could be suspended (due to priority-based policy) by higher priority threads. This is not a problem until such a higher-priority thread also wants to take the mutex. Being already acquired, the higher-priority thread must wait, being blocked (so, ceasing the CPU) and giving the mutex holder a change to run again. However, if other threads with priorities between the two mentioned before are running, they keep from running both the lower thread (mutex holder) and, indirectly, the higher thread waiting for the mutex. This looks like the priority rule was switched, a higher one waiting for a smaller one, like if there priorities would have been switched (this is why the problem is called priority inversion). This could lead to critical problems, when critical high-priority threads could be delayed indefinitely by smaller priority threads, so a good OS (like ours) tries to solve it. One solution to this problem is the priority (temporal) donation technique, which consists in the blocking high-priority thread donating its priority the the smaller-priority mutex holder, in order to boost it such that no in-between thread be able to delay the high one. Obviously, the donated priority must be kept only while the mutex holder has the mutex. When the mutex is released the donated priority must be lost and the thread should be restored to its real one. If this were the only possible case, the implementation of priority donation solution would be very simple. This is not the case however, as we will see below. However, it is clear from the problem definition that we have to handle the two priorities of a thread in the following functions:

1. `MutexAcquire()`, where priority donation could happen, and
2. `MutexRelease()`, where priority restoration must happen, if needed.

Basically, in `MutexAcquire()` we should do the following on the execution path corresponding to the mutex being held, when the current thread must

be blocked:

```
while (Mutex->Holder != pCurrentThread)
{
    crtThPrio = ThreadGetPriority(pCurrentThread);
    holderPrio = ThreadGetPriority(Mutex->Holder);

    if (crtPrio > HolderPrio)
    {
        // priority donation
        Mutex->Holder->Priority = crtThPrio;
    }
}
```

Similarly, in `MutexRelease()` we must restore the priority of mutex holder to its original value:

```
if (Mutex->Holder->Priority != Mutex->Holder->RealPriority)
{
    Mutex->Holder->Priority = Mutex->Holder->RealPriority;
}
```

In practice it is not so simple, though, because a thread could hold simultaneously multiple mutexes. In that case it could be donated multiple priorities (so, its effective priority being boosted multiple times), due to the multiple mutexes it held. In such a case, it would not be right to restore that thread's priority to its real one when releasing one of its acquired mutexes, as it could still held other ones. The correct solution should take into account both the thread's real priority and the ones donated due the mutexes still being held by that thread. The formula would be something like: $max(real\_priority, max(donated_priorities))$

The reason the real priority should be considered is that the thread could change its real priority to a higher value, after being donated a priority due to a mutex is held.

There are different strategies to keep track of donated priorities, but the one we considered is to maintain for each thread a list of the mutexes it holds. Noting that each mutex is associated a waiting list, i.e. a list of all thread waiting for that mutex, it is very simple to see which such threads have a greater priority than the mutex holder, and consider such a priority as a donation. This is the reason we defined the field `AcquiredMutexesList` for each thread. As with any variable, let us see how it is initialized and used:

1. `AcquiredMutexesList` initialization should be done at thread creation and we chose to do it in `_ThreadInit`;

2. inserting an element (i.e. a mutex) in that list must be done when a mutex is acquired by the thread, i.e in function `MutexAcquire()` on the execution path corresponding the the mutex being acquired (after the while loop, waiting for the mutex to become available);

3. removing an element (i.e. a mutex) from that list must be done when a mutex is no longer held by a thread, so in function `MutexRelease()`.

Because we must add `MUTEX` structures to the `AcquiredMutexesList`, we need to add a `LIST_ENTRY` field in that structures, which we name `AcquiredMutexListElem`. Using this field, steps 2 and 3 described above could be done like this: `InsertTailList(GetCurrentThread()->AcquiredMutexList, &Mutex->AcuiredMutexListElem);` for the former and `RemoveEntryList(&Mutex->AcuiredMutexListElem)` for the later. Take a look of the implementation of `InsertTailList()` and `RemoveEntryList()` functions to understand their functionality and required parameters.

The next logical step we must establish is the more complex way a thread's priority is recomputed, when that thread releases one of its held mutexes. So the very simple algorithm described above in function `MutexRelease()` was replaced by another one that we will place in function `ThreadRecomputePriority()`, which basically performs the following steps:

1. initialize a current maximum value to the thread's real priority;

2. iterate the `AcquiredMutexesList` of the thread;

3. for each mutex, iterate its waiting list;

4. for each waiting thread in that list, compares its effective priority to the current maximum, and if bigger, updates the maximum to the new value;

5. set the thread's effective priority `Priority` to the maximum value found.

We will always use the `ThreadGetPriority()` function to correctly get the thread's effective priority. Take care that while iterating a mutex's waiting list in Step 3 above, that list could be concurrently changed by a thread waiting for the corresponding mutex, so entering the waiting list of that mutex. In ordor to avoid race conditions, we must take the lock of the mutex while iterating through its waiting list.

Another problem described in relation to priority inversion is the case a thread holding a mutex is waited by a higher priority thread (which, as established, donates its priority to the mutex holder), but the waiting thread is in its turn the holder of another mutex, which could be later on waited by

an even higher-priority thread. The latter thread, would donate its priority to the second we mentioned, but if this donation would not go even further to the first (smallest) thread, the donation would not be effective. This is what lead to the nested priority requirements. However, in order to be able to manage it, we need to know for each thread, when being donated a priority (due to holding a mutex), if that thread is not waiting in its turn after another thread (holding another mutex, in its turn). In order to manage this, we decided to add a new field `WaitedMutex` in the `THREAD` structure, to keep a pointer to a mutex that is waited for by the thread. In case there is no such waited mutex, the pointer would be NULL. We must note that there is no need, in this case for a list of waited mutexes, while, the thread being a single, sequential execution, there is no way to wait for multiple mutexes in the same time (i.e. while being block waiting for a mutex, no execution of that thread take place, such that to be able to attempt taking another mutex).

The initialization to NULL of the field `WaitedMutex` will be done in `_ThreadInit`, while a valid value (i.e. a pointer to a real mutex) will be done when a thread is going to be blocked waiting for an already acquired mutex, which is in function `MutexAcquire()` just before calling `ThreadBlock()`.

The usage of the `WaitedMutex` must be done in `MutexAcquire` also, when a thread is going to be blocked, but in relation to the priority donation for the mutex holder. We want to place the priority donation algorithm in a function called `ThreadDonatePriority`, which will perform the following steps:

1. compare the donor's priority with the mutex holder's one, and if the former's in greater, performs the donation, i.e. assigns the mutex holder's effective priority to that of the donor's effective priority;

```
donor = GetCurrentThread();
if (ThreadGetPriority(donor) > ThreadGetPriority(MutexHolder))
{
  MutexHolder->Priority = donor->Priority;
}
```

2. if the mutex holder is in its turn waiting for another mutex, the donation must go further, like

```
MutexHolder->WaitedMutex->Holder->Priority = MutexHolder->Priorit
```

3. the same nested mechanism must be repeated until the end of the waiting chain, i.e. until a thread waiting for no mutex.

Finally, when a thread succeeds acquiring a mutex it waited for previosly, its `WaitedMutex` must be set to NULL again. This must be done on the

execution path corresponding the the mutex being acquired (after the while loop, waiting for the mutex to become available).

The last problem we have to deal with in relation to priority inversion is about a thread changing its own real priority, in function `ThreadSetPriority()`. In this case, it is possible for a thread to increase its priority to a value higher than its currently donated priority (i.e. its effective priority), in which case, obviously, the effective priority must be given the same value like the current real one. A very simple way (though, not so efficient) to compute the thread's new effective priority would be to call the function `ThreadRecomputePriority()`.

While changing the effective priority of a thread during the (nested) priority donation process, we can face race conditions, because there could be multiple donations taking place in the same time: for instance, multiple threads trying to get different mutexes, whose holders are all waiting for the same mutex, so there could be concurrent attempts to change the priority of that mutex holder. Also, if that thread wants to change its own priority, calling `ThreadSetPriority()` function, there is an additional competing attempt to change that thread's effective priority. Because of race conditions like these, a thread's priority fields must be protected by a lock. We add for this the `PriorityProtectionLock` lock.

3.3 Tests

Your are given in your *HAL9000* the tests your solution will be checked against and evaluated and you are not required to develop any addition test. Though, even if the tests are known, it would be helpful for you during the design phase to take a look at the given tests, because that way you can check if your design covers all of them. It would be sufficient for most of tests to just read the comments describing them.

In this section you have to list the tests affecting your design and given a short description of each one (using you own words).

Timer

- `TestThreadTimerAbsolute()`: Creates an absolute timer set to expire in 50 ms, calculated using `IomuGetSystemTimeUs()`. Through `ExTimerInit()`, `ExTimerStart()`, and a single `ExTimerWait()` call, the test verifies that the waiting thread resumes precisely when the absolute target time is reached.

- `TestThreadTimerAbsolutePassed()`: Sets an absolute timer to a timestamp that occurred 50 ms in the past. Since the deadline has already elapsed, `ExTimerWait()` must return immediately. This confirms that timers scheduled in the past are handled correctly and do not block the thread.

- `TestThreadTimerPeriodicMultiple()`: Configures a relative periodic timer with a 50 ms interval and waits on it ten times in succession. The thread calls `ExTimerWait()` after each expiration, ensuring that periodic wakeups occur at accurate intervals across all cycles.

- `TestThreadTimerPeriodicOnce()`: Uses the same periodic setup as the previous test but waits only for the first expiration. This isolates the validation of the initial periodic trigger, confirming that the thread resumes execution after a single 50 ms delay.

- `TestThreadTimerPeriodicZero()`: Starts a relative periodic timer with a zero period. Each call to `ExTimerWait()` should complete instantly, demonstrating that the thread is never blocked and continues execution immediately.

- `TestThreadTimerSharedTimer()`: All worker threads (typically 16) share one relative once timer initialized in `TestThreadTimerPrepare()`. Each thread waits in `ExTimerWait()`, and once the shared timer expires after 50 ms, all threads should resume simultaneously.

- `TestThreadTimerMultipleTimers()`: Assigns each thread its own relative periodic timer, with the period set to $50\,\text{ms} + 10\,\text{ms} \times i$ for thread i. The test checks that threads wake up in the correct order according to their individual timer intervals.

- `TestThreadTimerRelative()`: Creates a single relative once timer configured for a 50 ms timeout. The helper `_TestThreadSleepWaitTimer()` performs a single `ExTimerWait()` call to confirm that the thread resumes exactly after the expected delay.

- `TestThreadTimerRelativeZero()`: Programs a relative once timer with a zero-millisecond delay. The call to `ExTimerWait()` must return immediately, showing that a zero-duration timer never blocks the waiting thread.

Priority Scheduler. Fixed Priority Scheduler

- `TestThreadPriorityWakeup()`: Launches several threads (typically 16) whose priorities range incrementally from `ThreadPriorityLowest` upward. Each thread waits on an `EX_EVENT` object, and the purpose of the test is to confirm that the scheduler awakens the threads in order of descending priority; higher-priority threads should resume execution before lower-priority ones.

- `TestThreadPriorityMutex()`: Performs the same validation as `TestThreadPriorityWakeup()`, but replaces event synchronization with `MUTEX` objects. The test ensures that when multiple threads of varying priorities contend for a mutex, they acquire it according to their priority ranking.

- `TestThreadPriorityYield()`: Creates a single thread running at maximum priority and verifies that it remains continuously scheduled, even when it voluntarily calls `ThreadYield()`. The goal is to ensure that the scheduler does not preempt a top-priority thread that chooses to yield execution.

- `TestThreadPriorityRoundRobin()`: Starts multiple threads (by default 16) all assigned the same highest priority, and checks that invoking `ThreadYield()` causes the CPU to rotate fairly among them. This confirms that threads sharing the same priority are scheduled in a round-robin manner.

Priority Scheduler. Priority Donation

- `TestThreadPriorityDonationOnce()`: Tests if a higher priority thread can donate its priority to a different thread which has acquired a mutex it is waiting for.

- `TestThreadPriorityDonationLower()`: Follows the same structure as the previous test, however it checks that the thread which received the donation maintains it even when trying to lower its own priority.

- `TestThreadPriorityDonationMulti()`: The main thread creates two mutexes which it holds initially, and two threads, A and B, which try to acquire those mutexes. After thread B is allowed to execute, the test checks if the main thread still has A's priority

- `TestThreadPriorityDonationMultiInverted()`: Same as the previous test, however the mutexes are released in the inverse order. The main thread should have B's priority.
- `TestThreadPriorityDonationMultiThreads()`: Same as the previous test, however with three threads, A, B, and C. B and C have a higher priority, and are both waiting for the second mutex. After the second mutex is released, the main thread should have a priority equal to B/C's.
- `TestThreadPriorityDonationMultiThreadsInverted()`: Same as the previous test, however the mutexes are released in the inverse order. The main thread should have B/C's priority.
- `TestThreadPriorityDonationNest()`: Tests if nested threads can succesively donate their priorities.
- `TestThreadPriorityDonationChain()`: Same as the previous test, however with more threads.

3.4 Observations

It was an interesting subject to work on. It required much time than I estimated. I learned that a good design is not a trivial thing to do and requires some time to be done.

Chapter 4

Design of Module *Userprog*

4.1 Assignment Requirements

4.1.1 Initial Functionality

Argument Passing

The existing HAL9000 implementation provides the essential functionalities for thread and process management needed to run user programs. However, it currently does not implement support for parsing program arguments and does not provide a mechanism to push those arguments onto the stack.

Validation of System Call Arguments

When launching user programs that take arguments, the system only performs a simple validation to see if the buffer is usable or not. Because of this, invalid or unchecked input may cause the operating system to crash. Access-rights checking should also be extended to cover multiple pages.

System Calls for Process Management

The fundamental logic for starting and ending processes is already provided by the kernel (such as the functions `ProcessCreate()` and `ProcessTerminate()`). However, the starting point only has a functional implementation for `SyscallProcessExit()` and nothing for closing a handle or creating, waiting for, or obtaining a process's ID.

System Calls for File System Access

The starting point contains a partial implementation for the `SyscallFileWrite()` function, without anything regarding file handling. We also do not have other functions regarding creating, reading and closing a file.

System Calls for Thread Management

The current HAL9000 operating system provides only kernel-level functionality and lacks any meaningful interface for user programs to make requests from the operating system. User applications cannot perform essentialy basic operations such as accessing files, creating processes, or managing threads.

4.1.2 Requirements

Argument Passing

Before starting a program, its command-line arguments must be pushed onto the user stack. This requires supporting multiple arguments, storing them as a null-terminated array of strings, and keeping the stack correctly aligned.

Validation of System Call Arguments

Here we must verify that all arguments passed by a user during a system call are valid. This includes checking that pointers and buffers are in user space, confirming buffer sizes to avoid overflows or illegal memory access, and ensuring the user has the necessary access rights.

System Calls for Process Management

To enable user programs to generate, manage, and control processes within the operating system, system calls are necessary for process management. Creating processes (`SyscallProcessCreate()`), waiting for processes to end (`SyscallProcessWaitForTermination()`), ending processes (`SyscallProcessExit()`), obtaining process identifiers (`SyscallProcessGetPid()`), and closing process handles (`SyscallProcessCloseHandle()`) are all included in this. These system calls are crucial for controlling the lifetime of processes, guaranteeing correct execution, synchronization, and cleanup. To avoid resource leaks and preserve system stability, the implementation must also securely handle process-specific data, manage process isolation, and check inputs.

System Calls for File System Access

System calls are necessary for file system access so that user programs can safely and effectively interact with files. This entails implementing functions for reading data from files (`SyscallFileRead()`), closing file handles (`SyscallFileClose()`), and creating or opening files (`SyscallFileCreate()`). By managing file activities while maintaining data integrity, these system calls guarantee restricted access to file resources. To avoid resource leakage and guarantee system dependability, the implementation must also check inputs, preserve segregated access to file resources, and guarantee appropriate file handle management.

System Calls for Thread Management

Implement the system calls `SyscallThreadExit()`, `SyscallThreadCreate()`, `SyscallThreadGetTid()`, `SyscallThreadWaitForTermination()` and `SyscallThreadCloseHandle()`.

4.1.3 Basic Use Cases

Argument Passing

With argument passing, users can type input in the command line to change how a program runs. It lets them set different values or options without editing the program itself.

Validation of System Call Arguments

The purpose of the validation is to protect the kernel from malicious attempts to access restricted memory or exploit buffer overflows that could reach kernel space

System Calls for Process Management

It may be necessary for a user application to run a process that performs a specific task, such as generating code or carrying out a computation, and then obtain the result once the process has finished. To do this, the application starts a new process by calling `SyscallProcessCreate()`. After creation, it may call `SyscallProcessGetPid()` to retrieve the process identifier for tracking or logging purposes. The program then uses `SyscallProcessWaitForTermination()` to wait until the process completes, while the running process itself calls `SyscallProcessExit()` to signal that it has finished execution. Finally, once

the process ends, the application calls `SyscallProcessCloseHandle()` to release the handle and free any associated resources, ensuring proper cleanup and efficient process management

System Calls for File System Access

In order to load configuration settings or process user input, a user application may need to read data from a file. In order to accomplish this, the program will open or create the file by calling `SyscallFileCreate()`. The application will use `SyscallFileRead()` to retrieve data from the file when it has been opened. This may involve reading binary data or lines of text. For writing, the application will call `SyscallFileWrite()`. In the end, the application will call `SyscallFileClose()` to shut the file and release the related resources after the read process is complete. This guarantees effective resource management and correct file access.

System Calls for Thread Management

File Editing: A basic text editor requires access to the file system in order to create and update files. Various essential commands such as `ls`, `touch`, and `rm` require access to the file system.
Build System: A compilation tool like `make` needs to compile multiple source files in parallel to reduce build time. It uses system calls to spawn compiler processes for independent files, passes command-line arguments specifying input files and compiler flags, and uses wait operations to synchronize completion before linking.

4.2 Design Description

4.2.1 Needed Data Structures and Functions

Argument Passing

Functions that need to be updated: In `thread.c` we will modify:

- `ThreadSetupMainThreadUserStack()`: This function must be updated so it can extract the arguments from the `PROCESS` structure and correctly push them onto the `USER STACK` in the proper order.

We also need to introduce a new data structure to manage the user stack. This will give us a clean and organized way to interact with the user-mode stack.

```
struct _USER_STACK
{
    PVOID   StackPointer;
    QWORD   StackSize;
    char**  argv;
    QWORD   argc;
    PVOID   Reserved;
} USER_STACK, *PUSER_STACK;
```

Since multiple data types will be pushed onto the stack, and we want the code to remain readable instead of placing everything in one location, it is useful to create helper functions that push different types of values onto the user stack. We will implement the following helper functions:

- `StackPushString()`: pushes a null-terminated string onto the user stack.
- `StackPushInt()`: pushes an integer value onto the user stack.
- `StackPushPointer()`: stores a pointer value onto the user stack.

Validation of System Call Arguments

In this section, we will extend the behavior of some existing functions and introduce new logic to validate arguments passed to a system call.

Functions to be updated:

- `MmuIsBufferValid()`: currently verifies buffer validity, but must be enhanced to check all memory pages that the buffer spans.
- `CmmIsBufferValid()`: must also be modified so it iterates through every page occupied by the buffer.

System Calls for Process Management

Handle Model (per-process)

We use **per-process** handle tables for *process*, *thread*, and *file* objects. A handle is a small integer returned to user mode; the kernel validates it on every system call and checks that it belongs to the **current process** and matches the **expected type**. This keeps isolation simple.

Data structures (added headers). We add the following definitions to a new header `headers/um_handle_manager.h` and include it where needed. Embed the table into `PROCESS` (see below).

```
typedef enum _HANDLE_TYPE {
    THREAD_HANDLE,
    PROCESS_HANDLE,
    FILE_HANDLE,
    INVALID_HANDLE
} HANDLE_TYPE;

typedef struct _UM_HANDLE_ENTRY {
    DWORD       HandleId;     // unique in owning process
    HANDLE_TYPE Type;         // expected object class
    PVOID       Object;       // PTHREAD / PPROCESS / PFILE_OBJECT
    LONG        RefCount;     // for safe close
    LIST_ENTRY  ListEntry;    // link inside per-process table
} UM_HANDLE_ENTRY, *PUM_HANDLE_ENTRY;

typedef struct _UM_HANDLE_TABLE {
    LOCK        Lock;         // protects the table
    LIST_ENTRY  Head;         // all UM_HANDLE_ENTRY items
    DWORD       NextHandleId; // starts at 1
} UM_HANDLE_TABLE, *PUM_HANDLE_TABLE;
```

Member in PROCESS (added field).

We add this to `headers/executive/process_internal.h` inside the `PROCESS` structure:

```
UM_HANDLE_TABLE HandleTable;    // per-process handle table
```

Minimal handle manager (new source).

Provide a small manager with the signatures below. These are thin list/lock helpers used by all syscalls.

```
STATUS UmHandleInitTable(PUM_HANDLE_TABLE T);
void UmHandleUninitTable(PUM_HANDLE_TABLE T);
STATUS UmHandleCreate(PUM_HANDLE_TABLE T, HANDLE_TYPE Type, PVOID Obj, DWORD* Out);
STATUS UmHandleResolve(PUM_HANDLE_TABLE T, DWORD H, HANDLE_TYPE Expect, PVOID* OutObj);
void UmHandleClose(PUM_HANDLE_TABLE T, DWORD H);
```

Functions for System Calls for Process Management

Process syscalls create, resolve, and close *per-process* handles. On any invalid handle, null/invalid pointer, or type mismatch, the handler returns an error.

- `SyscallHandler()`: we update this function to include a new case in the switch statement for each newly implemented system call.
- `SyscallProcessCreate()`: creates a new process that executes the code specified by the `FilePath` parameter, using the provided `Arguments`.
- `SyscallProcessWaitForTermination()`: blocks the calling thread until the process identified by `ProcessHandle` has finished execution.
- `SyscallProcessGetPid()`: retrieves the unique process identifier (PID) associated with a given process.
- `SyscallProcessCloseHandle()`: closes the process handle specified by `ProcessHandle`, releasing any associated resources.

Functions for System Calls for File System Access

File syscalls follow the same per-process handle pattern and use the kernel's I/O entry points.

- `SyscallFileCreate()`: creates a new file in the file system.
- `SyscallFileRead()`: reads the contents of a file and stores the data in a buffer for later processing.
- `SyscallFileClose()`: closes an open file and releases the associated system resources.

System Calls for Thread Management

Create the following new functions

- `SyscallThreadExit(Status)`: Exist the currently running thread and handles resource cleanup. Notifies any waiting threads. Returns an exit status to the operating system.
- `SyscallThreadCreate(ThreadHandle, Routine, Arg)`: Create a new thread inside the current running process. The thread will begin executing *Routine* using the given *Arg*. A handle will be returned inside the *ThreadHandle* pointer. Returns *0* on success.

- `SyscallThreadGetTid(ThreadHandle, ThreadId)`: Get the TID for the given *ThreadHandle* and store it inside *ThreadId*.
- `SyscallThreadWaitForTermination(ThreadHandle, Retval)`: Wait for the termination of *ThreadHandle*. Its return value will be stored inside the *Retval* pointer.
- `SyscallThreadCloseHandle(ThreadHandle)`: Close the handle associated with *ThreadHandle*

System Calls for Thread Management

Create the following new functions

- `SyscallThreadExit(Status)`: Exist the currently running thread and handles resource cleanup. Notifies any waiting threads. Returns an exit status to the operating system.
- `SyscallThreadCreate(ThreadHandle, Routine, Arg)`: Create a new thread inside the current running process. The thread will begin executing *Routine* using the given *Arg*. A handle will be returned inside the *ThreadHandle* pointer. Returns *0* on success.
- `SyscallThreadGetTid(ThreadHandle, ThreadId)`: Get the TID for the given *ThreadHandle* and store it inside *ThreadId*.
- `SyscallThreadWaitForTermination(ThreadHandle, Retval)`: Wait for the termination of *ThreadHandle*. Its return value will be stored inside the *Retval* pointer.
- `SyscallThreadCloseHandle(ThreadHandle)`: Close the handle associated with *ThreadHandle*

4.2.2 Interfaces Between Components

Argument Passing and Validation of System Call Arguments

Pointer Validation: Argument passing introduces a function to verify pointers supplied by user programs. This function ensures that arguments passed from user space reference valid, properly aligned memory regions.Thread and process management system calls reuse this pointer validation logic to check user-provided pointers such as thread entry addresses, context values,and other parameters.

Memory Safety: The memory validation logic developed for argument passing ensures safe interaction with user memory, preventing kernel crashes or privilege violations. This functionality is also used in thread management when working with user-space arguments or initializing new thread contexts.

System Calls for Process Management

The system call handlers for process management interact with the kernel functions for process management by invoking functions defined in `process.h` and `process.c`, such as `ProcessCreate()`, `ProcessWaitForTermination()`, `ProcessGetId()`, and `ProcessCloseHandle()`.

- `SyscallProcessCreate()` will use `ProcessCreate()` to create a new user-mode process from the specified executable file.
- `SyscallProcessWaitForTermination()` will use `ProcessWaitForTermination()` to suspend the current thread until the target process has exited.
- `SyscallProcessGetPid()` will use `ProcessGetId()` to obtain the unique identifier (PID) of a process.
- `SyscallProcessCloseHandle()` will use `ProcessCloseHandle()` to release the reference to a process resource.
- `SyscallProcessExit()` will use the existing `ProcessTerminate()` function to stop the currently running process.

System Calls for File System Access

The system call handlers for file system access interface with the I/O Manager by calling functions defined in `io.h` and `io_files.c`, such as `IoCreateFile()`, `IoReadFile()`, and `IoCloseFile()`.

- `SyscallFileCreate()` will use `IoCreateFile()` to create a new file when requested by a user process.
- `SyscallFileRead()` will use `IoReadFile()` to read data from a file and store it into a buffer for later processing.
- `SyscallFileClose()` will use `IoCloseFile()` to close an open file and release the associated data and resources.

System Calls for Thread Management

Handle Management:

Processes and threads will use the same data structure implementation for managing handles. Processes will use a global structure, while threads will use a per-process structrue.

A process must ensure that all its threads are cleaned up, before terminating itself.

Synchronization:

Processes and threads will both use kernel events for synchronization.

4.2.3 Analysis and Detailed Functionality

Design Strategy (Argument Passing)

Goal. Argument passing prepares a fresh user stack so that `main(int argc, char** argv)` sees the expected inputs. In HAL9000 this work is driven by `_ThreadSetupMainThreadUserStack()`: it parses the command line, copies the strings to the user stack, builds the `argv` pointer array, and finally places `argc` and `argv` on the stack with the proper ABI alignment.

Inputs and context. The function reads the raw command line and argument count from the `PROCESS` object (`FullCommandLine` and `NumberOfArguments`). Because the kernel cannot write directly into a new process's stack, it acquires a temporary kernel mapping with `MmuGetSystemVirtualAddressForUserBuffer()` and releases it with `MmuFreeSystemVirtualAddressForUserBuffer()` once setup is complete.

Working state. We track stack construction through `_USER_STACK` (fields: `StackPointer`, `StackSize`, `argc`, `argv`, and `Reserved`). As bytes and pointers are pushed, `StackPointer` is updated so that layout, alignment, and shadow space are honored.

Placement order. Strings are copied first (last argument to first) using `OnStackPushString()` so their bytes become contiguous at lower addresses. Then we emit the `argv` array by pushing the addresses of those strings in reverse order via `OnStackPushPointer()`. Finally, we place `argc` with

`OnStackPushInt()` and the `argv` base pointer with `OnStackPushPointer()`, after enforcing stack alignment and reserving any required shadow space and a dummy return address.

```
// Build initial user-mode stack for the process's main thread.
// Returns STATUS and the resulting user-mode stack pointer.
FUNCTION _ThreadSetupMainThreadUserStack(
    IN  PVOID     InitialStack,
    OUT PVOID*    ResultingStack,
    IN  PPROCESS  Process
) -> STATUS
{
    STATUS     status = SUCCESS
    USER_STACK userStack = {0}
    PVOID      KernelStackAddress = NULL

    // Initialize tracking state.
    userStack.StackPointer = InitialStack
    userStack.StackSize    = DEFAULT_STACK_SIZE
    userStack.argv         = NULL
    userStack.argc         = Process->NumberOfArguments
    userStack.Reserved     = NULL

    // Map the user stack into kernel space so we can write to it.
    KernelStackAddress =
        MmuGetSystemVirtualAddressForUserBuffer(
            userStack.StackPointer,
            userStack.StackSize,
            WRITE_ACCESS,
            Process)

    IF (KernelStackAddress == NULL) THEN
        RETURN MEMORY_ERROR
    ENDIF

    // Parse the command line into tokens (equivalent to using strchr/strtok).
    ARG_LIST args = SplitCommandLine(Process->FullCommandLine)

    // 1) Copy argument strings onto the stack, from last to first.
    //    Record each string's user-mode address for argv[].
    VECTOR<PVOID> ArgAddrs = EMPTY
    FOR i FROM args.count-1 DOWNTO 0 DO
        PVOID strUserAddr = NULL
```

```
    userStack.StackPointer =
        OnStackPushString(
            KernelStackAddress,
            userStack.StackPointer,
            args[i],
            &strUserAddr)
    ArgAddrs.PUSH_FRONT(strUserAddr)   // keep argv[0]..argv[n-1]
END FOR

// 2) Enforce ABI alignment (e.g., 16-byte).
userStack.StackPointer =
    AlignDown(userStack.StackPointer, ABI_STACK_ALIGN)

// 3) Reserve shadow space if required by the calling convention.
userStack.StackPointer =
    ReserveShadowSpace(KernelStackAddress,
                       userStack.StackPointer,
                       SHADOW_SPACE_BYTES)

// 4) Build argv[] by pushing pointers in reverse order.
FOR j FROM ArgAddrs.count-1 DOWNTO 0 DO
    userStack.StackPointer =
        OnStackPushPointer(
            KernelStackAddress,
            userStack.StackPointer,
            ArgAddrs[j])
END FOR
PVOID argvBase = UserAddressOf(KernelStackAddress, userStack.StackPointer)

// 5) Push argc and argv.
userStack.StackPointer =
    OnStackPushInt(KernelStackAddress, userStack.StackPointer, (UINT64)userStack.arg
userStack.StackPointer =
    OnStackPushPointer(KernelStackAddress, userStack.StackPointer, argvBase)

// 6) Push a synthetic return address (convention-specific).
userStack.StackPointer =
    OnStackPushPointer(KernelStackAddress, userStack.StackPointer, FAKE_RETURN_ADDRE

// Done: return final SP and drop the temporary mapping.
*ResultingStack = userStack.StackPointer
MmuFreeSystemVirtualAddressForUserBuffer(KernelStackAddress)
RETURN status
```

```
}

//Helpers

// Push a null-terminated string; return updated SP and out its user address.
FUNCTION OnStackPushString(
    IN  PVOID  KernelStackAddress,
    IN  PVOID  StackPointer,
    IN  STRING String,
    OUT PVOID* OutUserAddress
) -> PVOID
{
    SIZE len = ByteLen(String) + 1        // include '\0'
    SIZE padded = AlignUp(len, 8)
    PVOID newSP = (PVOID)((UINTPTR)StackPointer - padded)

    MEMCOPY((BYTE*)KernelStackAddress + OffsetOf(newSP), String, len)
    MEMSET ((BYTE*)KernelStackAddress + OffsetOf(newSP) + len, 0, padded - len)

    *OutUserAddress = UserAddressOf(KernelStackAddress, newSP)
    RETURN newSP
}

FUNCTION OnStackPushInt(
    IN PVOID KernelStackAddress,
    IN PVOID StackPointer,
    IN UINT64 Value
) -> PVOID
{
    PVOID newSP = (PVOID)((UINTPTR)StackPointer - SIZEOF(UINT64))
    WRITE64((BYTE*)KernelStackAddress + OffsetOf(newSP), Value)
    RETURN newSP
}

FUNCTION OnStackPushPointer(
    IN PVOID KernelStackAddress,
    IN PVOID StackPointer,
    IN PVOID Pointer
) -> PVOID
{
    PVOID newSP = (PVOID)((UINTPTR)StackPointer - SIZEOF(PVOID))
    WRITEPTR((BYTE*)KernelStackAddress + OffsetOf(newSP), Pointer)
    RETURN newSP
```

```
}

// Reserve raw bytes (e.g., shadow space or padding) and return the new SP.
FUNCTION ReserveShadowSpace(
    IN PVOID KernelStackAddress,
    IN PVOID StackPointer,
    IN SIZE  Bytes
) -> PVOID
{
    (void)KernelStackAddress  // not used; kept for symmetry
    RETURN (PVOID)((UINTPTR)StackPointer - AlignUp(Bytes, ABI_STACK_ALIGN))
}
```

System Call Argument Validation

Overview. Before the kernel can safely serve a system call, every argument supplied by a user process must be validated. The OS receives pointers and buffers from user space, and it must confirm that each one points to accessible, valid, and properly permissioned memory. Without strict checking, a malicious or faulty program could crash the kernel or access privileged memory.

Current State. HAL9000 already performs basic checks: it ensures the buffer is not `NULL`, verifies that it does not point into kernel space, and confirms access permissions for the region. These checks are handled via `MmuIsBufferValid()`, which in turn calls into `VmmIsBufferValid()` and `VmReservationReturnRightsI`

However, the current implementation only evaluates the first region of memory. If a buffer spans more than one page or crosses reservation boundaries, only the initial portion is checked, which leaves open a possibility for unsafe memory access.

Required Improvement. To strengthen validation, the system must iterate through all pages covered by the buffer. Each page may hold different rights, so every segment must pass security checks. This guarantees that a buffer partially mapping to valid memory and partially into protected memory cannot slip through.

Permission and String Handling. The validation function must also handle strings carefully. When a string buffer is provided, it must:

- end with a `\0` byte,

- remain within its allocated region,
- not run into unmapped memory.

If the string is not properly terminated or exceeds its bounds, the system call should fail immediately to prevent memory corruption or leaks.

Execution Flow. The enhanced `MmuIsBufferValid()` performs initial pointer checks, verifies that the address belongs to user space, determines how many pages the buffer covers, and walks through each one using `VmmIsBufferValid()`. If any checked segment fails, the function exits with an error and the system call does not continue.

This approach ensures full memory validation without unnecessary overhead, maintaining both correctness and system security.

```
FUNCTION MmuIsBufferValid(Buffer, BufferSize, RightsRequested, Process) -> STATUS
{
    // Reject any kernel memory access attempt
    IF Buffer IN KERNEL_SPACE THEN
        RETURN STATUS_MEMORY_PREVENTS_USERMODE_ACCESS
    ENDIF

    // Validate each page touched by the buffer
    FOR page FROM Buffer TO (Buffer + BufferSize) STEP PAGE_SIZE DO
        status = VmmIsBufferValid(
                    page,
                    PAGE_SIZE,
                    RightsRequested,
                    Process.VaSpace,
                    Process.PagingData.KernelSpace)
        IF status IS NOT SUCCESS THEN
            RETURN status
        ENDIF
    END FOR

    // Final full-range permission verification
    status = VmmIsBufferValid(
                Buffer,
                BufferSize,
                RightsRequested,
                Process.VaSpace,
                Process.PagingData.KernelSpace)
    IF status IS NOT SUCCESS THEN
        RETURN status
```

```
    ENDIF

    // Additional checks for string buffers (read access)
    IF RightsRequested INCLUDES READ_ACCESS THEN
        hasNull = FALSE

        FOR offset FROM 0 TO BufferSize DO
            IF *(Buffer + offset) == '\0' THEN
                hasNull = TRUE
                BREAK
            ENDIF
        END FOR

        IF hasNull == FALSE THEN
            RETURN STATUS_STRING_TOO_LONG   // no terminator found
        ENDIF
    ENDIF

    RETURN STATUS_SUCCESS
}
```

System Calls for Process Management

• `SyscallProcessCreate()`
We first validate all user-supplied pointers (path, arguments, and output handle) across page boundaries and copy the strings into kernel memory. If the path is relative, we normalize it to the Applications folder on the system drive. We then ask the kernel loader to create the process, yielding a `PPROCESS` on success. This object is registered in the **current process's** handle table by creating a new integer handle, associating it with the kernel object, and marking its type as `PROCESS`. The new handle is written back to user mode; any validation, normalization, or loader error is returned as a status code.
• `SyscallProcessWaitForTermination()`
We validate the output status pointer and then **resolve** the supplied handle in the **current process's** handle table under lock. We verify that the handle exists and that its type is `PROCESS`. After successful resolution, we use the kernel's wait mechanism on the target process's termination event, which blocks the calling thread until the process exits. We then obtain the final exit code and copy it back to the user. If the handle is not owned by the caller, is invalid, or has the wrong type, we fail the call without waiting.
• `SyscallProcessGetPid()`

There are two modalities. If the user passes the sentinel `INVALID` handle, we return the PID of the **current** process from the process hierarchy. Otherwise, we resolve the given handle in the **current process's** handle table and verify that its type is `PROCESS`. If resolution and type check succeed, we obtain the PID of the target process and return it to the user; otherwise the call fails.

- `SyscallProcessCloseHandle()`

A user closes a process handle they no longer need. We locate and resolve this handle in the **current process's** handle table, verify its type is `PROCESS`, then remove it from the table. We dereference the underlying kernel object so the kernel can reclaim it when the last reference is gone. If the handle is missing, not owned by the caller, or of the wrong type, we return an error.

System Calls for File System Access

- `SyscallFileCreate()`

We validate the path and output handle pointers and copy the path into kernel memory. For a relative path, we interpret it relative to the system drive root. The user-mode stub enters the kernel, where the `SyscallHandler` invokes the internal `IoCreateFile` to open or create the file/directory and return a kernel file object on success. We then create a new integer handle in the **current process's** handle table, associate it with the file object, mark the type as `FILE`, and return this handle to user mode. Validation, normalization, or open/create failures are returned as status codes.

- `SyscallFileRead()`

We validate the output buffer and `BytesRead` pointers and check the requested size. We resolve the supplied handle in the **current process's** handle table and verify that its type is `FILE`. If resolution succeeds, the kernel reads up to the requested byte count into the user buffer using safe copy helpers and updates `BytesRead` with the actual number of bytes read. Invalid, non-owned, or mistyped handles cause the call to fail.

- `SyscallFileClose()`

We pass the file handle to the kernel and remove it from the **current process's** handle table after verifying it is valid and of type `FILE`. We decrement the file object's reference count; when no references remain, the kernel closes the file and releases its resources. If the handle is invalid, not owned by the caller, or has the wrong type, the close fails with an error.

System Calls for Thread Management

Thread Management

```
SyscallThreadExit (
   IN STATUS Status
)
```

The SyscallThreadExit allows a thread to gracefully terminate its execution. The Status will be forwarded to other threads waiting for this termination. Firstly, we must swith to kernel mode and then call ThreadExit using the SyscallHandler. Inside ThreadExit we must set the Status and signal to other threads waiting. The handle associated with this thread is removed from the process' handle table, and finally, the execution is passed to a new thread by calling the scheduler.

```
SyscallThreadCreate(
   OUT    UM_HANDLE* ThreadHandle,
   IN     PFUNC_ThreadStart Routine,
   IN_OPT PVOID Arg
)
```

The SyscallThreadCreate allows for the creation of a new thread within the current running process. A Routine and Arg are given, and a ThreadHandle is returned.

Simmilarly to the previous syscall, we first switch to kernel mode, before calling ThreadCreateEx. After creating the thread, we generate a new handle and store it within the current running process.

```
SyscallThreadGetTid(
   IN_OPT UM_HANDLE ThreadHandle,
   OUT    TID* ThreadId
)
```

Retrieve the Thread ID associated to the given ThreadHandle. If ThreadHandle is not provided, return the TID of the current running thread.

After switching to kernel mode, iterate through the process' handle list until finding the handle we are searching for. Afterwards, return the associated TID.

```
SyscallThreadWaitForTermination(
   IN     UM_HANDLE ThreadHandle,
   OUT    STATUS* Retval
)
```

The SyscallThreadWaitForTermination allows for synchronization between threads. The currently running thread will pause its execution until the thread associated with ThreadHandle terminates. The return value is stored inside Retval.

After switching to kernel mode, use synchronization primitives, such as `ExEvenWaitForSignal` to block the current thread until `ThreadHandle` finished execution. Afterwards, copy the target thread's `Status` inside `Retval`.

```
SyscallThreadCloseHandle(
    IN      UM_HANDLE ThreadHandle,
)
```

Clean up thread handles and release all associated resources if no thread is waiting for `ThreadHandle`.

Switch to kernel mode, find `ThreadHandle` inside the process' handle list, and decrement the reference count. If the reference count reaches zero, call `_ThreadDestroy` and remove the handle from the list.

4.3 Tests

Argument Passing Tests

- `TestUserArgsNone`: Ensures a program runs correctly with no arguments. In this case, `argc = 0` and `argv` includes only a single `NULL` entry.
- `TestUserArgsOne`: Validates proper handling of a single argument. `argc = 1` and `argv` contains two entries: `argv[0]` pointing to the argument string, and `argv[1] = NULL`.
- `TestUserArgsMany`: Checks correct processing of several arguments, verifying that parsing and population of the `argv` array are handled accurately.
- `TestUserArgsAll`: Executes a stress test by supplying a large number of arguments, confirming stable and correct argument handling under heavy input conditions.

Validation of System Call Arguments - Bad Actions

These tests verify that the OS rejects illegal memory accesses and unauthorized privileged operations from user programs.

- `BadJumpKernel`: Checks system behavior if a user attempts to execute code in kernel memory.

- `BadJumpNoncanonical`: Ensures an attempted jump to a non-canonical address triggers the correct fault.
- `BadJumpNull`: Verifies system response when execution is attempted at a `NULL` pointer.
- `BadReadIoPort`: Confirms that I/O privilege restrictions are enforced by attempting to read from I/O ports.
- `BadReadKernel`: Ensures user code cannot read from kernel memory.
- `BadReadMsr`: Validates that reading from privileged MSR registers is blocked.
- `BadReadNonCanonical`: Tests behavior when reading from a non-canonical virtual address.
- `BadReadNull`: Checks that reading from `NULL` is properly handled.
- `BadWriteIoPort`: Verifies that writing to protected I/O ports is blocked.
- `BadWriteKernel`: Ensures attempts to write into kernel memory fail correctly.
- `BadWriteMsr`: Confirms that writing to MSR registers is disallowed.
- `BadWriteNonCanonical`: Ensures invalid writes to non-canonical addresses trigger a fault.
- `BadWriteNull`: Validates system handling of writes to `NULL` memory.

System Calls for Process Management

These tests validate the correctness and robustness of process management system calls and the handle management subsystem.

- `ProcessCloseFile`: Validates type safety. Attempts to close a valid file handle using `SyscallProcessCloseHandle()`. The system must detect the wrong handle type and return an error.
- `ProcessCloseNormal`: Confirms that a valid process handle can be successfully closed, properly decrementing the kernel object's reference count.
- `ProcessCloseParentHandle`: Ensures that a child process cannot close a handle belonging to its parent.

- `ProcessCloseTwice`: Checks robustness. The first close should succeed, and a second close on the same handle must fail.
- `ProcessCreateBadPointer`: Validates argument safety by calling `SyscallProcessCreate()` with an invalid pointer. The kernel must fail safely without crashing.
- `ProcessCreateMissingFile`: Attempts to create a process from a non-existent executable file. The system must return a "file not found" error.
- `ProcessCreateMultiple`: Stress test that spawns several processes simultaneously to confirm stability and handle safety under load.
- `ProcessCreateOnce`: Basic successful process creation from a valid executable.
- `ProcessCreateWithArguments`: Ensures command-line arguments are correctly passed to the new process.
- `ProcessExit`: Runs a child process that calls `SyscallProcessExit()` with a specific status. The parent must receive the correct exit code.
- `ProcessGetPid`: Verifies that `SyscallProcessGetPid()` returns the correct PID for both the current and target processes.
- `ProcessWaitBadHandle`: Calls `SyscallProcessWaitForTermination()` with an invalid handle; must return an error.
- `ProcessWaitClosedHandle`: Waits on a closed process handle; must fail, confirming correct handle invalidation.
- `ProcessWaitNormal`: Parent process waits until the child exits and resumes correctly afterward.
- `ProcessWaitTerminated`: Ensures that waiting on an already-terminated process returns immediately with the correct exit code.

System Calls for File System Access

These tests validate file system operations and handle management logic specific to file objects.

- `FileCloseBad`: Attempts to close an invalid file handle; must fail.

- `FileCloseNormal`: Confirms that a valid file handle can be closed successfully.
- `FileCloseStdout`: Attempts to close `STDOUT_FILENO`; must be explicitly disallowed.
- `FileCloseTwice`: First close succeeds; second must fail.
- `FileCreateBadPointer`: Calls `SyscallFileCreate()` with a NULL filename pointer; must fail safely.
- `FileCreateEmptyPath`: Attempts to create a file with an empty string path; must fail.
- `FileCreateExistent`: Creates a file that already exists without overwrite permission; must fail.
- `FileCreateMissing`: Tries to open (not create) a non-existent file; must return "file not found".
- `FileCreateNormal`: Successfully creates a new empty file; the standard valid case.
- `FileCreateNull`: Equivalent to a NULL path test; must fail.
- `FileCreateTwice`: Attempts to create the same file twice; the second attempt must fail.
- `FileReadBadHandle`: Reads using an invalid handle; must fail.
- `FileReadBadPointer`: Calls `SyscallFileRead()` with an invalid buffer pointer; must fail safely.
- `FileReadKernel`: Attempts to read into kernel memory; must be detected and rejected.
- `FileReadNormal`: Performs a valid read from a known file and verifies the data.
- `FileReadStdout`: Attempts to read from write-only `STDOUT_FILENO`; must fail.
- `FileReadZero`: Requests to read zero bytes; must succeed and report zero bytes read.

System Calls for Thread Management

- `ThreadCloseTwice`: Try to close the same thread handle twice. The second call to close the handle should fail.
- `ThreadCreateBadPointer`: Try to create a thread and pass an invalid function pointer. The process should crash.
- `ThreadCreateMutliple`: Creates a set amount of threads (10) and waits for their termination. All threads should be created and exited successfully, and had their handles closed with no errors.
- `ThreadCreateOnce`: Create a single thread and close its handle afterwards.
- `ThreadCreateWithArguments`: Create a thread and pass arguments to the starting function. Compare the received arguments with the expected arguments and terminate the thread successfully.
- `ThreadExit`: Test the functionality of the `SyscallThreadExit`. The execution should stop immediately.
- `ThreadGetTid`: Spawn threads and validate the uniqueness and consistency of their TIDs accross multiple threads.
- `ThreadWaitBadHandle`: Try waiting a thread on a bad handle. The call should fail.
- `ThreadWaitClosedHandle`: Try waiting a thread on a closed handle. The call should fail.
- `ThreadWaitNormal`: Test the behaviour of `SyscallThreadWaitForTermination` under normal circumstances. The call should succeed and the exit status shoud match with that provided by the target thread.
- `ThreadWaitTerminated`: Try waiting for a thread after it has already terminated. Call to `SyscallThreadWaitForTermination` should return successfully.

4.4 Observations

It was an interesting assignment to work on. We have learnt how the user programs interact with and run on the OS.

Chapter 5

Design of Module *virtualmemory*

5.1 Assignment Requirements

5.1.1 Initial Functionality

Per Process Quotas

The processes are currently allowed to open as many files as they want and also use as many physical frames as they want.

Zero Pages

The OS doesn't allow allocating pages which are zero-initialized. Any attempt to do so will result in an assertion failure.

5.1.2 Requirements

Per Process Quotas

The processes are limited (each one) to a maximum number of 16 open files and 16 physical frames. If a process attempts to exceed these limits, it will be prevented. In the case of files, it will be prevented from opening another one until one is closed, and in the case of physical frames, one of them will be swapped to the disk.

Zero Pages

For implementing zero pages allocation, a change has to be done in `VmmAllocRegionEx()` in which the zero allocation assertion has to be removed and the function

has to be updated with an implementation which allocates pages with zero bytes.

5.1.3 Basic Use Cases

Per Process Quotas

Process quotas limit how many resources a process can use, which results in fair allocation and prevention of system instability. By applying these rules, the OS ensures that no process can overwhelm its resources.

Zero Pages

Zero Pages are used for performance optimization in the OS. They allow the apps to allocate memory pages which are initialized to zero, thus resulting in clean and secure memory allocation. It is very helpful when the apps need predictable or sanitized memory for working.

5.2 Design Description

5.2.1 Needed Data Structures and Functions

Per Process Quotas

```
typedef struct _PROCESS
{
    ...
    DWORD FramesNo;
    DWORD FilesNo;
} PROCESS, *PPROCESS;
```

These 2 fields are kept in each process and they count the number of currently opened files and used frames. They are used to prevent a process for exceeding the limit of 16 opened files simultaneously and 16 physical frames. Updated functions:

- `SyscallFileCreate()`: will increment the `FilesNo` field. If the limit is reached, further openings will be prevented until a file is closed.
- `SyscallFileClose()`: will decrement the `FilesNo` when an open file is closed.

- `SyscallVirtualAlloc()`: will increment the `FramesNo` field. If the limit is reached, the swapping mechanism is triggered to move a frame to the disk.
- `SyscallVirtualFree()`: will decrement the `FramesNo` field when a frame is freed.

Also we may create an error status if the quota is exceeded:
`#define STATUS_QUOTA_EXCEEDED ((STATUS)0xC0000044L)`

Zero Pages

The function `VmmAllocRegionEx()` will be modified to remove the assertion `ASSERT(!IsBooleanFlagOn(AllocType, VMM_ALLOC_TYPE_ZERO));` . This will be replaced with an implementation which initializes the allocated memory to zero by using the `memzero()` function.

5.2.2 Interfaces Between Components

Per Process Quotas and Zero Pages

They interact with the function `VmmSolvePageFault()`. It happens when a process exceeds the number of allowed physical frames (the field `PROCESS_MAX_PHYSICAL_FRAMES` in the file `process_internal.h`). The components also interact with the system calls to the virtual memory system management and the stack growth by keeping track of the number of allocated frames.

5.2.3 Analysis and Detailed Functionality

Per Process Quotas

```
STATUS
SyscallFileCreate(
    IN_READS_Z(PathLength) char*   Path,
    IN QWORD                      PathLength,
    IN BOOLEAN                    Directory,
    IN BOOLEAN                    Create,
    OUT UM_HANDLE*                FileHandle
)
{
    ...
    currentProcess = GetCurrentProcess();
```

```
    // ---> Check the quota before trying to open a new file <---
    if (currentProcess->FilesNo >= PROCESS_MAX_OPEN_FILES)
    {
        return STATUS_QUOTA_EXCEEDED;
    }
    ...

    if (!SUCCEEDED(status))
    {
        IoCloseFile(fileObj);
        return status;
    }

    // ---> Increment the number of opened files <---
    AtomicIncrement32((volatile DWORD*)&currentProcess->FilesNo);
    // we include the atomic functions from "cal_atomic.h"

    *FileHandle = (UM_HANDLE)handle;

    return STATUS_SUCCESS;
}
```

```
STATUS
SyscallFileClose(
    IN UM_HANDLE FileHandle
)
{
    ...
    IoCloseFile(fileObj);

    // --> Decrement the file counter <---
    if (currentProcess->FilesNo > 0)
    {
        AtomicDecrement32((volatile DWORD*)&currentProcess->FilesNo);
    }

    return STATUS_SUCCESS;
}
```

```
STATUS
SyscallVirtualAlloc(
    IN_OPT      PVOID                   BaseAddress,
    IN          QWORD                   Size,
```

```
    IN          VMM_ALLOC_TYPE          AllocType,
    IN          PAGE_RIGHTS             PageRights,
    IN_OPT      UM_HANDLE               FileHandle,
    IN_OPT      QWORD                   Key,
    OUT         PVOID*                  AllocatedAddress
)
{
    ...
    framesNeeded = (DWORD)(
        AlignAddressUpper(Size, PAGE_SIZE) / PAGE_SIZE
    );

    // ---> Check Frame Quota <---
    if (
        currentProcess->FramesNo + framesNeeded >
        PROCESS_MAX_PHYSICAL_FRAMES
    )
    {
        // This is where Swapping would be triggered
        return STATUS_QUOTA_EXCEEDED;
    }

    // Call internal VMM
    resultAddress = VmmAllocRegionEx(
        BaseAddress,
        Size,
        (VMM_ALLOC_TYPE)AllocType,
        (PAGE_RIGHTS)PageRights,
        FALSE,
        NULL,
        currentProcess->VaSpace,
        currentProcess->PagingData,
        NULL
    );
    if (resultAddress == NULL)
    {
        return STATUS_MEMORY_CANNOT_BE_COMMITED;
    }

    // ---> Update Frame Quota <---
    AtomicExchangeAdd32(
        (volatile DWORD*)&currentProcess->FramesNo,
        framesNeeded
```

```
    );

    ....
}
```

```
STATUS
SyscallVirtualFree(
    IN PVOID BaseAddress,
    IN QWORD Size,
    IN DWORD FreeType
)
{
    ...
    VmmFreeRegionEx(
        BaseAddress,
        Size,
        (VMM_FREE_TYPE)FreeType,
        TRUE,
        currentProcess->VaSpace,
        currentProcess->PagingData
    );

    // ---> Decrement Frame Quota <---
    if (currentProcess->FramesNo >= framesFreed)
    {
        AtomicExchangeAdd32(
            (volatile DWORD*)&currentProcess->FramesNo,
            -((INT32)framesFreed)
        );
    }
    else
    {
        currentProcess->FramesNo = 0;
    }

    ...
}
```

Zero Pages

```
PTR_SUCCESS
PVOID
```

```
VmmAllocRegionEx(
    IN_OPT  PVOID                   BaseAddress,
    IN      QWORD                   Size,
    IN      VMM_ALLOC_TYPE          AllocType,
    IN      PAGE_RIGHTS             Rights,
    IN      BOOLEAN                 Uncacheable,
    IN_OPT  PFILE_OBJECT            FileObject,
    IN_OPT  PVMM_RESERVATION_SPACE  VaSpace,
    IN_OPT  PPAGING_LOCK_DATA       PagingData,
    IN_OPT  PMDL                    Mdl
    )
{
    ...
    ASSERT(Mdl == NULL ||
           IsBooleanFlagOn(AllocType, VMM_ALLOC_TYPE_NOT_LAZY)
    );

    // ---> Support zero pages initialization <---
    //ASSERT(!IsBooleanFlagOn(AllocType, VMM_ALLOC_TYPE_ZERO));
    ...

    if (FileObject != NULL)
    {
        QWORD fileOffset;
        QWORD bytesRead;

        // make sure we read from the start of the file
        fileOffset = 0;

        status = IoReadFile(FileObject,
                            alignedSize,
                            &fileOffset,
                            pBaseAddress,
                            &bytesRead);
        if (!SUCCEEDED(status))
        {
            LOG_FUNC_ERROR("IoReadFile", status);
            __leave;
        }

        LOG_TRACE_VMM(
            "File size is 0x%X, bytes read are 0x%X\n",
            alignedSize,
```

```
            bytesRead
        );

        // zero the rest of the file
        ASSERT(alignedSize - bytesRead <= MAX_DWORD);

        // memzero the rest of the allocation
        // (in case the size of the allocation is greater than the
        // size of the file)
        memzero(
            PtrOffset(pBaseAddress, bytesRead),
            (DWORD)(alignedSize - bytesRead)
        );
    }
    // ---> Support zero pages initialization <---
    else if (IsBooleanFlagOn(AllocType, VMM_ALLOC_TYPE_ZERO))
    {
        memzero(pBaseAddress, alignedSize);
    }
    ...
}
```

5.3 Tests

Per Process Quotas

- `ProcessQuotaGood()`: this test checks if the process can open and close a maximum number of 16 files without exceeding the limit. It keeps on opening and closing files, checking whether or not the OS properly handles the limits imposed and allows reopening files after they've been closed

- `ProcessQuotaJustRight()`: this test checks if the process can open exactly 16 files (the limit). It checks whether or not no files are left open after they're closed, seeing if the OS correctly handles the file count

- `ProcessQuotaMore()`: this test checks if the OS allows the process to open more than 16 files, by attempting to open 32. It expects failure when the limit is exceeded

Zero Pages, together with Swapping

- `SwapLinear()`: tests that the swapping method works smoothly for basic memory functions. It starts by creating a big block of memory and writing data in a precise pattern to each page. The test then checks if the data remains correct and consistent, even when some of the pages are changed out to disk and eventually brought back into physical memory. To further check the functioning, it substitutes the written data with zeros and checks that the memory accurately reflects the changes. This test covers the essential parts of swapping, like evicting pages, writing them to the swap partition, recovering them as needed, and con- firming that the memory contents are exactly as expected after both the data and zero-write operations.

- `SwapMultiple()`: evaluates the swapping mechanism's dependability in a multi-process setting. It involves numerous processes allocating and accessing memory at the same time, placing the system under stress to handle simultaneous swapping activities. The test checks that the system stays stable and that each process can only access its own allocated memory, even when pages are switched out to disk and later swapped back into physical memory. To properly validate the system's stability, this operation is performed four times, confirming that the swapping capability functions consistently under repeated and concurrent memory demands.

- `SwapMultipleShared()`: This test takes the multi-process swapping scenario a step further by providing shared memory between processes. It starts by allocating a 16MB block of shared memory, then spawns multiple child processes. Each child writes data to a specific portion of the shared memory, using a set pattern identical to earlier testing. Once all the child processes complete their tasks, the main process performs a thorough check to ensure the entire memory block is correct, while the child processes verify the bits they wrote to. This test ensures that shared memory is consistent and available to all processes, even when the shared pages are swapped to disk and brought back into physical memory.

- `SwapZeros()`: This test makes sure that during the swapping operation, memory initialized to zero remains consistent. It starts by allocating a 256MB block of memory using the `VMM_ALLOC_TYPE_ZERO` flag, which guarantees that the memory is initially filled with zeros. The test then checks each page in the block to confirm that all values remain zero.

If any page contains a non-zero value, it triggers an error. This test is designed to verify that zero-initialized memory is properly handled, continues intact after being switched out to disk and back into physical memory, and avoids any corruption or unexpected changes along the way.

- `SwapZerosWritten()`: By guaranteeing that any modifications made to zero-initialized memory are maintained throughout the swapping procedure, this test builds upon the `SwapZeros` test. As in the previous test, it starts by allocating memory initialized to zero and verifying that the first 16MB are all zeros. Following that, particular values are written to the remaining memory, and the test confirms that these values are appropriately stored. When pages are switched out to disk and then brought back into physical memory, this procedure guarantees that the swapping system treats altered memory consistently, preserving any modifications. The test confirms the system's capacity to retain data integrity and appropriately reflect updates made to memory, even under swapping situations.

5.4 Observations

It was an interesting and very useful assignment to work on. We have learnt how the virtual memory is managed on an OS.

Chapter 6

Design of Module *Processor-Affinity and FS Cache*

6.1 Assignment Requirements

6.1.1 Initial Functionality

Describe in few words (one phrase) what you are starting from in your project. Even if this is something that we all know, it could be a good opportunity for you to be sure you really understand this aspect.

6.1.2 Requirements

Remove the following given official requirements and describe in few words (your own words) what you are requested to do in this part of your project. Even if this is something that we all know, it could be a good opportunity for you to be sure you really understand this aspect.

The requirements of the "Processor Affinity" assignment are the following:

1. *On-Processor Ready Lists.* You must place ready threads in per-processor ready lists and change the thread scheduler to support scheduling threads from those lists. You must implement a kind of "pull-based" processor balancing, which means pulling (i.e. scheduling) threads on an idle processor from the ready list of another processor. Deciding which such a list to use could be based on a policy you decide, e.g. a random non-empty ready list, the ready lists with the maximum number of threads etc. You must deal with possible race conditions implied by your algorithm. You are not allowed to use a single global lock to synchronize

concurrent instances of the scheduler running simultaneously on different CPUs anytime they try to choose a ready thread, especially when the ready list of one CPU is not empty. Centralized synchronization strategy (and maybe the usage of one lock) could be activated only when a general balancing of all ready lists is needed, if your scheduling policy decides so.

2. *Processor Affinity.* You should provide support for user processes to inform the OS about the processors one of their threads should (or must) be run on, a property named "processor affinity". A more detailed explanation of what processor affinity is could be found here.

   (a) You must add an additional optional parameter of type QWORD to the thread creation system call, which will function as a processor-affinity bitmap. Bits with value 1 in the processor-affinity parameter indicate processors on which the thread could be run, while 0 values indicate processors on which the thread could not be run. The least significant bit (i.e .bit 0) correspond to the logical processor having the identifier 0, the second least significant bit (i.e. bit 1) to the logical processor with identifier 1 and so on. For example, if the processor-affinity parameter of one thread would be "`0xFF`" (i.e. "`0b1111'1111`"), that thread could run on any of the first 8 processors. If the the processor-affinity parameter would be "`0x06`" (i.e. "`0b00000110`") it could be run only on processors with identifier 1 and 2. You could use for processor identification the `PCPU.LogicalApicId` field, associating it to a particular bit in the processor-affinity mask. We recommend you building a per-system processor mask at system initialization, based on the available number of processors and their `LogicalApicId`. For instance, if you run your OS on a system with just two logical processors having their `LogicalApicId` 1 and 4 respectively, you processor mask would be `0x05`. In such a case, if a thread affinity would be expressed as `0xFF` (i.e. all processors), you should filter out the unavailable processors, getting the effective affinity-mask `0x05`.

   (b) Implement a system call "ThreadSetProcessorAffinity" to change the calling thread current affinity-mask to the value specified by the system call argument. For a detailed description of such a system call read this document. When a thread's processor-affinity is changed, the scheduler should rescheduled that thread if it is

currently running on a processor not included anymore in that thread's processor-affinity mask.

3. *File-System Cache.* Modify the provided FAT file system to keep a cache of file blocks. When a request is made to read or write a block, check to see if it is in the cache, and if so, use the cached data without going to disk. Otherwise, fetch the block from disk into the cache, evicting an older entry if necessary.

   You are limited to a cache no greater than 64 sectors in size. You must implement a cache replacement algorithm that is at least as good as the "clock" (i.e. second-chance) algorithm. We encourage you to account for the generally greater value of metadata compared to data. Experiment to see what combination of accessed, dirty, and other information results in the best performance, as measured by the number of disk accesses.

   You can keep a cached copy of the FAT table permanently in memory if you like. It doesn't have to count against the cache size.

   Your cache should be *write-behind*, that is, keep dirty blocks in the cache, instead of immediately writing modified data to disk. Write dirty blocks to disk whenever they are evicted. Because write-behind makes your file system more fragile in the face of crashes, in addition you should periodically write all dirty, cached blocks back to disk. The cache should also be written back to disk when the system halts.

   You should also implement *read-ahead*, that is, automatically fetch the next block of a file into the cache when one block of a file is read, in case that block is about to be read. Read-ahead is only really useful when done asynchronously. That means, if a process requests disk block 1 from the file, it should block until disk block 1 is read in, but once that read is complete, control should return to the process immediately. The read-ahead request for disk block 2 should be handled asynchronously, in the background.

The way to allocate requirements on member teams.

- 3-members teams

  1. Per-Processor Ready Lists
  2. Processor Affinity
  3. File-System Cache

6.1.3 Basic Use Cases

Try to describe a real-life situation, where the requested OS functionality could be useful in a user-application or for a human being. This is also an opportunity for you to better understand what the requirements are and what are they good for. A simple use-case could be enough, if you cannot find more or do not have enough time to describe them.

6.2 Design Description

6.2.1 Needed Data Structures and Functions

This should be an overview of needed data structure and functions you have to use or add for satisfying the requirements. How the mentioned data structures and functions would be used, must be described in the next subsection "Detailed Functionality".

6.2.2 Interfaces Between Components

In this section you must describe the identified interference of your component(s) with the other components (existing or developed by you) in the project. You do not have to get in many details (which go into the next section), but must specify the possible inter-component interactions and specify the existing functions you must use or existing functions you propose for handling such interactions.

6.2.3 Analysis and Detailed Functionality

Here is where you must describe detailed of your design strategy, like the way the mentioned data structures are used, the way the mentioned functions are implemented and the implied algorithms.

This must be the main and the most consistent part of your design document.

It very important to have a coherent and clear story (text) here, yet do not forget to put it, when the case in a technical form. So, for instance, when you want to describe an algorithm or the steps a function must take, it would be of real help for your design reader (understand your teacher) to see it as a pseudo-code (see an example below) or at least as an enumerated list. This way, could be easier to see the implied steps and their order, so to better understand your proposed solution.

6.2.4 Explanation of Your Design Decisions

This section is needed, only if you feel extra explanations could be useful in relation to your designed solution. For instance, if you had more alternative, but you chose one of them (which you described in the previous sections), here is where you can explain the reasons of your choice (which could be performance, algorithm complexity, time restrictions or simply your personal preference for the chosen solution). Though, try to keep it short.

If you had no extra explanation, this section could be omitted at all.

6.3 Tests

Your are given in your *HAL9000* the tests your solution will be checked against and evaluated and you are not required to develop any addition test. Though, even if the tests are known, it would be helpful for you during the design phase to take a look at the given tests, because that way you can check if your design covers all of them. It would be sufficient for most of tests to just read the comments describing them.

In this section you have to list the tests affecting your design and given a short description of each one (using you own words).

6.4 Observations

This section is also optional and it is here where you can give your teacher a feedback regarding your design activity.