The Design of *Pintos* Operating System

Petra Mariel, Pintilei Ovidiu, Trif Gheorghe Andrei

14 octombrie 2020

Rezumat

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.

Capitolul 1

General Presentation

1.1 Working Team

Specify the working team members' names and their responsibility.

1. Firstname1 Familyname1
   (a) Threads: dealt with AAA
   (b) Userprog: dealt with BBB
   (c) VM: dealt with CCC
2. Firstname2 Familyname2
   (a) Threads: dealt with AAA
   (b) Userprog: dealt with BBB
   (c) VM: dealt with CCC
3. Firstname3 Familyname3
   (a) Threads: dealt with AAA
   (b) Userprog: dealt with BBB
   (c) VM: dealt with CCC

Capitolul 2

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. *design* 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 Pintos 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 our 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 Pintos project because 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 Pintos structure and, on the other hand, that you cannot make your design ignoring the Pintos' code. Even if, theoretically, a design could be abstract enough to support different implementations (consequently, containing no particular code), your Pintos design has to make direct references to some existing data structures and functions, when they are needed for the functionality of the Pintos 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 "*struct thread*" data structure. Or, when you need to keep a list of some elements, you have to use the lists already provided by Pintos and show that by declaring and defining your list in the way Pintos does, like below:

```
struct list my_list;           // this is the way Pintos
                               // declare a generic list

struct my_list_elem {
    ... some fields ...

    struct list_elem my_list_elem;  // this is neeed for
                    // linking the my_list_elem in the list
                    // while Pintos implements generic lists
};
```

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 Pintos project. In the Pintos 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 Pintos 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 be 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 Pintos 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 Pintos code in the "tests/" subdirectory. Take at least a short reading of each test comments 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. *Pintos* 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 *Pintos* 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 Pintos 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.

```
struct existent_data_structure {
    ...
    int newField;
    char newField2;
    ...
};

struct newDataStructure {
    ... new fields ...
};
```

```
int newFunction();
```

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.

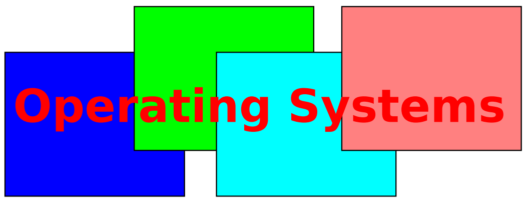

Figura 2.1: Sample image

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 2.3. 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
```

IMPORTANT NOTE: For each module's design template (see the following chapters) you are provided some questions that you have to answer in order to check if your design covers some special aspects and situations. Obviously, you can read that questions before making your design, just to find out some aspects you have to deal with. However, do not organize this subsection (i.e. Section "2.3.2") strictly based on that questions just answering them one by one. MAKE YOU DESIGN DESCRIPTION A LOGICAL AND NICE STORY! Not a Q&A section. The answers of the given questions should be given implicitly in your description, not necessarily asking the questions themselves. It could be though acceptable to answer explicitly each question in turn, but only at the end of your personal description.

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

A very simple way to find out which are the names of the tests is to execute the command below in the directory corresponding to each assignment:

```
make check
```

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 *Pintos* development, starting from the design phase.

In this section you have to describe briefly each of the given *Pintos* 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 Pintos tests.

2.5 Observations

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

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

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

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

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

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

Capitolul 3

Design of Module *Threads*

3.1 Assignment Requirements

3.1.1 Initial Functionality

Alarm Clock

We are given an implementation of an alarm clock (see file "`devices/timerc.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 `next_thread_to_run()` in file `threads/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 `Schedule()` in file `threads/thread.c` and `switch_threads` in file `threads/switch.S`).

Priority Scheduler. Priority Donation

Same given code as described at previous section.

Advanced Scheduler: Multi-Level Feedback Queue Scheduler (MLFQ))

Same given code as described at previous section.

3.1.2 Requirements

Alarm Clock

We must change the given alarm clock's functionality, such that to not use anymore the busy waiting technique in function `timer_sleep()`, but a blocking mechanism, which suspends the waiting thread (i.e. takes the CPU from it) until the waited alarm clock expires. The blocking / unblocking technique template we could use is already applied in the semaphore implementation, yet you must adapt it to our alarm clock's implementation.

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.

Advanced Scheduler: Multi-Level Feedback Queue Scheduler (MLFQ))

We must implement a priority-based scheduler, but differently by the one describe at Section 3.1.2, we must establish thread priority dynamically, during thread runtime, based on some given criteria (like time consumed for running, i.e. using, on the CPU, time spent waiting for a CPU or for some other event). The required algorithm is called multi-level feedback queue (MLFQ), suggesting the continuous increase or decrease of thread priorities, based on their runtime behavior. The rules to change the threads' priorities are described in the Pintos documentation.

3.1.3 Basic Use Cases

Alarm Clock

A user application could use an alarm clock 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.

Advanced Scheduler: Multi-Level Feedback Queue Scheduler (MLFQ))

This kind of functionality is also almost invisible from user space and there is no explicit way to control or test it from there. It generally could be tested by observing the way the interactive threads in user applications react to user input. Such threads are blocked most of the time, waiting for use input (e.g. keyboard taste, mouse click), but it is very important to be given a priority boost when awaken, such that to be able to rapidly get the CPU to give the user a fast response. This could only be obtain if MLFQ scheduling is used and the priority changing criteria are well tuned.

3.2 Design Description

3.2.1 Needed Data Structures and Functions

Alarm Clock

We will add in `timer.h` the structure `struct timer`:

```
struct alarm_clock
{
    // time to wake up
    int64_t             waking_time;

    // thread to be awaken
    struct thread       *thread;

    // to add the timer in the global list of timers
    struct list_elem    alarm_clock_elem;
};
```

We add in `timer.c` a global list keeping track of all timers in the system:

```
static struct list alarm_clock_list;
```

Functions that we will change:

- `timer_sleep()`: replace the busy-waiting with the blocking technique;
- `thread_tick()`: call the function that check which timers must be triggered and their corresponding threads awaken (i.e unblocked);

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

- `alarm_clock_compare(struct list_elem t1, struct list_elem t2, void* aux)`: compare two timer trigger time in order to keep the global timer list order by timer triggering time;
- `alarm_clock_check(struct timer* t)`: called on each timer interrupt handling to check for a particular alarm clock if it must be triggered or not;

- `alarm_clock_check_all(void)`: called on each timer interrupt handling to check for all alarm clocks in the global alarm clock 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 `struct thread` structure, we must only use it.

```
struct thread
    ...

    int priority;

    ...
};
```

We will change the following functions:

- `thread_unblock()`: to insert the unblocked thread in the ready list such that to keep it ordered descending by thread priority and to check if the current thread must be preempted or not;

- `thread_yield()`: to insert the current thread giving up the CPU in the ready list such that to keep it ordered descending by thread priority; this way, it is possible to have the same thread being given again the CPU, while being the highest-priority thread in ready list;

- `sema_down()`: to insert the blocking thread in the semaphores' waiting list such that to keep it ordered descending by thread priority;

- `cond_wait()`: to insert the semaphore on which the calling therad is blocked in the condition's list such that to keep it ordered descending by thread priority;

- `thread_set_priority()`: to check if changing the calling thread's priority requires or not the current thread to give up the CPU (in case its priority is decreased) or not.

The new functions that we will add are:

- `thread_compare(struct list_elem e1, struct list_elem e2, void* aux)`: to compare two threads by their priority;
- `cond_compare(struct list_elem e1, struct list_elem e2, void* aux)`: to compare two condition variables by their waiting threads' priority.

Priority Scheduler. Priority Donation

Priority donation supposes giving a thread a new temporal priority, while that thread is holding a lock. 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.h`:

```
struct thread
{
    int priority;       // the effective priority (already defined)

    ...

    // Used for priority donation
    int         real_priority;          // the real (original) priority
    struct list acquired_locks_list;    // the list of locks held by thre
    struct lock* waited_lock;          // the lock thread waits for
    ...
}
```

In file `synch.h`:

```
struct lock
    ...

    unsigned value;              /* Current value. */
    struct list waiters;         /* List of waiting threads. */

    // elem in list of locks acquired by a thread
    struct list_elem acquired_lock_list_elem;
    ...
};
```

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

- `lock_acquire()`: donate priority of blocking thread (in case the mutex in already acquired), if its priority is higher than that of lock holder;
- `lock_release()`: recompute the priority of the thread releasing the lock;
- `thread_set_priority()`: take into account both real priority, which is changed by the functions, and the current effective (donated) one;
- `thread_get_priority()`: 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:

- `thread_donate_priority()`: called in `lock_acquire()` to donate priority to a lock holder and also deal with the nested donation aspect;
- `thread_recompute_priority()`: called in `lock_release()` to recompute the priority of a thread just releasing a lock, taking into account its real priority, but also priorities donated due to other locks that thread still holds.

Advanced Scheduler: Multi-Level Feedback Queue Scheduler (MLFQ))

To be determined by students. Read the given documentation, while all the needed details are described there.

3.2.2 Interfaces Between Components

All 3 tasks are related to each other so in our view, to be able to implement the fixed priority scheduler and the priority donation we must first implement the alarm clock because the timer_sleep function is needed in this scheduler. Another interference is that in the implementation of the fixed priority scheduler, when we have to schedule the threads with the same priorities we must use the round-robin technique which will take the threads in the order they were added to the ready-list. So we must reuse some of the existing code. Another interference is that to make a fully implemented priority donation scheduler we must first implement perfectly the fixed priority scheduler. So in our opinion the steps of the implementation are alarm clock, fixed priority scheduler and then priority donation.

3.2.3 Analysis and Detailed Functionality

Alarm Clock

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 `thread_block()` (in file `thread.c`) provides such a functionality. We took a look of places `thread_block()` was already used, to see the way the blocking mechanism is used. One good example was in function `sema_down()`, where we noted that before blocking the current thread (i.e. the one waiting for the semaphore), it was inserted in a waiting list, associated to that semaphore. We also noted that the blocked thread was unblocked in function `sema_up()`, by calling the function `thread_unblock()` on the blocked thread removed from the waiting list. So, our idea was to create a similar list where to keep threads waiting for an alarm clock to expire (i.e. for a specific time to be reached), using the `thread_block()` function to block a thread and `thread_unblock()` to unblock it. That list would be a global list.

In order to know for each thread waiting in that list the time it must be unblocked (i.e. the time its alarm clock "rings"), we will create in `timer.h` a new data structure, called `struct alarm_clock` (illustrated in Section 3.2.1), with the following fields:

1. `waking_time`: for the time moment the thread must be awaken, i.e. the alarm clock expires;
2. `thread`: to have a reference (pointer) to the thread associated to that alarm clock;
3. `alarm_clock_elem`: to be able to add the data structure in the global alarm clock list (see in `list.h` the way the generic lists are manages and used).

The global alarm clock list will declared in `timer.c`, like below (we declar it static, taking the model of `ready_list` in `thread.c`, just to make it visible only in file `timer.c` from modularity reasons):

```
static struct list alarm_clock_list;
```

Once we had established the needed data structures, we had to determine three aspects related to their usage:

1. when and how to create and initialize them;
2. where and how to use them, such that to block a thread waiting for an alarm clock to expire;
3. where and how to use them, such that to unblock the waiting threads, when their alarm clocks expire.

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

1. initialize the `alarm_clock_list` in function `timer_init()` in file `timer.c`, while it looks to be where the timer mechanism is initialized; the timer mechanisms is used to keep track of time passage in *Pintos*, based on the programmed timer interrupt;

2. create and initialize a new alarm_clock structure, when a thread calls the timer_sleep(), while there is where a thread wants to wait for a specific time to pass (i.e. an alarm clock to expire);

   - the waking_time field is initialized to the current system time (obtained by calling timer_ticks()) added to the time the thread wants to waits for (i.e. the timer_sleep function's parameter);
   - the thread field is initialized to the current thread, i.e. the one wanting to wait;

3. add the created structure to the global alarm clock list;

4. traverse the global alarm clock list to check which threads must be signaled, and remove from the list the alarm clock structures for expired clocks.

One problem we noted related to adding a new alarm clock structure in the global list is that this operation could happen concurrently in more threads. This suggests us that we must synchronize the access to that list. One idea was to use a lock for this. However, looking at how thread_block() was used in sema_down() we noted that another synchronization mechanism was used: disable the interrupts until the access to the waiting list was performed. Actually, disabling interrupts not only synchronize the access to the waiting list, but also links together the list insertion operation with the following blocking operation. Function thread_block() even checks, using an ASSERT, if the interrupts were disabled before being called. This is why we decided to use the same mechanism in function timer_sleep().

Another aspect we had to clarify was where to wake up (unblock) threads, when their alarm clock expires. Looking at the semaphore implementation, this was done in function sema_up(). Though, in our case, was not so obvious which was the equivalent function. The idea was to find out a function where we could check if the condition for an alarm clock to expire was fulfilled. This should be a function where we could measure the passage of time, which we found to be the function timer_interrupt(), called each time a timer interrupt occurred (it is the timer interrupt handling routine).

Based on the previous observations, we establish the steps that must be performed for implementing the required functionality.

The `timer_sleep()` function will perform the following steps:

1. create a new `struct alarm_clock` structure and initialize it as described above;
2. disable the interrupts;
3. add the data structure to the global alarm clock list, by calling `list_push_back()`;
4. block the current thread by calling the `thread_block()` function;
5. set the interrupts to their previous state (i.e. possible enable the interrupts);

In function `timer_interrupt()` we simply call the function `alarm_clock_check_all()`, which:

1. iterate the global alarm clock list and
2. call the `alarm_clock_check()` on each element in the list.

The `alarm_clock_check()` do the following:

1. compares the alarm clock's time to the current system time;
2. if the alarm clock's time is less than or equal to the current system time
   (a) remove the alarm clock's structure from the global list (by calling `list_pop_front()`), and
   (b) call `thread_unblock()` on the corresponding thread.

Disabling the interrupts in the `timer_interrupt()` is not needed, while they are already so, as they are in any interrupt routine.

An optimization we could apply on the way the global alarm clock list is managed could be to order ascending that list based on alarm clock expiration time. This way, when we iterate through the list checking which alarm clocks must be triggered, we only have to go until we find the first alarm clock with the triggering time bigger than the current system time. All the next elements in the list being even bigger, we could interrupt the list iteration. In order to do this, we must call `list_insert_ordered()` in `timer_sleep()` instead of `list_push_back()`. The `list_insert_ordered()` function requires an argument, which must be a comparison function. We will create for this the function `alarm_clock_compare(struct list_elem t1, struct list_elem t2, void* aux)`, which will perform the following:

1. obtains from its first two parameters, which are of the generic type `struct list_elem`, two pointers to `struct alarm_clock` structure, using the `list_entry` macro;

2. compares the two alarm clocks' wake-up times, returning TRUE if the first is less than second, and FALSE otherwise.

Priority Scheduler. Fixed-Priority Scheduler

The priority-based scheduler's policy requires that at any moment the highest priority thread (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 the currently running thread must preempt that 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 global variable `ready_list` in file `thread.c`);

2. semaphores' lists (see field waiters of structure struct sema, file synch.h); while locks are implemented using semaphores, ordering the semaphores' waiting lists, implies ordering the locks' waiting lists also;

3. condition variables' lists (see field waiters of structure struct cond, file synch.h).

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

- in thread_unblock() for inserting the unblocked thread in ready_list ordered descending by priority;
- in thread_yield() for inserting the yielding thread (i.e. the one giving up the CPU) in ready_list ordered descending by priority;
- in sema_down() for inserting the blocking thread (i.e. the one waiting for the semaphore or lock) in waiters ordered descending by priority;
- in cond_wait() for inserting the blocking thread (i.e. the one waiting for the condition variable), actually inserting the associated semaphore, in waiters ordered descending by priority.

The thread_compare() function works the following way:

1. obtains from its first parameter e1, which is of the generic type struct list_elem, a pointer to a struct thread structure, using the list_entry macro, whose the first parameter is the list element, the second is the struct thread structure containing that list element, and the third is the name of the list element field in the thread structure, which is the "elem":

   ```
   struct thread *pTh1;
   pTh1 = list_elem(e1, struct thread, elem);
   ```

2. similarly, obtains from its second parameter e2 a pointer to a struct thread structure:

```
struct thread *pTh2;
pTh2 = list_elem(e2, struct thread, elem);
```

3. compare the two threads' priorities, returning TRUE if the first is greater than or equal to the second, and FALSE otherwise, such that to order the list in a descendant way.

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

compare_and_return_result(prio1, prio2);
```

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

In order to order the condition variable's waiting queue decreasingly based on the waiting threads' priorities, we will implement a new function, called `cond_compare(struct list_elem e1, struct list_elem e2, void* aux)`, which will perform the following steps (considering that the list consist in semaphores, not threads, still in the waiting list of each semaphore there is a waiting thread):

1. obtains from its first parameter `e1`, which is of the generic type `struct list_elem`, a pointer to a `struct semaphore_elem` structure, using the `list_entry` macro, whose the first parameter is the list element, the second is the `struct semaphore_elem` structure containing that list element, and the third is the name of the list element field in the thread structure, which is the "`elem`". From the obtained semaphore's list extract the first (and single) element and convert it to the `struct thread`:

```
struct semaphore_elem *pSem1;
struct thread * pTh1;
pSem1 = list_elem(e1, struct semaphore_elem, elem);
pTh1 = list_elem(list_front(&pSem1->waiters), struct thread, elem);
```

2. similarly, obtains from its second parameter `e2` a pointer to a `struct semaphore_elem` structure and then a pointer to a `struct thread`:

   ```
   struct semaphore_elem *pSem2;
   struct thread * pTh2;
   pSem2 = list_elem(e2, struct semaphore_elem, elem);
   pTh2 = list_elem(list_front(&pSem2->waiters), struct thread, elem);
   ```

3. compare the two threads' priorities, returning TRUE if the first is greater than or equal to the second, and FALSE otherwise, such that to order the list in a descendant way.

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

   compare_and_return_result(prio1, prio2);
   ```

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 *Pintos*), which would take the threads in the order they were added to the ready list and will give them the CPU 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 `list_insert_ordered()` 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 `list_insert_ordered()` 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 the CPU, besides the existing one. The threads competing for the CPU are the running thread and the threads in the ready list, having the running thread with a higher priority than all those in ready list. However, a newly arrived thread could have a higher priority than the one running at that moment and this is why our scheduler should be called to preempt the running thread, 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 `thread_create()`;
2. an unblocked thread (see function `thread_unblock()`).

However, if when we looked in more details at function `thread_create()` we noted that a newly create thread is set active (i.e. ready) by calling the `thread_unblock()` 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 function `sema_up()`, where `thread_unblock()` is called). So, we concluded that the `thread_unblock()` 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, current_thread
Ensure: Preempt the running thread if smaller than the unblocked one
  $new\_prio$ = THREAD_GET_PRIORITY(unblocked_thread)
  $crt\_prio$ = THREAD_GET_PRIORITY(current_thread)
  if $new\_prio > crt\_prio$ then
      thread_yield()                    ▷ Preempt the currently running thread
  elseLIST_INSERT_ORDERED(ready_list, unblocked_thread)
  end if
```

Even if in our propose preemption mechanism we called `thread_yield()` only in the case the unblocked thread was higher that the currently running one, we must take care that `thread_yield()` is also called periodically in `thread_tick()`, due to the RR scheduling policy (still used for threads with the same priority, as described above). This means that it could be called, even when the currently running thread is the highest priority ready thread. In that case out `thread_yield()` implementation must assure that the current thread remains on the CPU. However, if we looked at our propose implementation of `thread_yield()`, while it inserts the current thread in the ready list and than calls the `schedule()` function, which chooses (by calling `next_thread_to_run()`) the first thread in the ready list, this is exactly the current thread, if it is the highest in the ready queue. This means that the current thread is actually not taken the CPU while being the highest in ready list.

We mentioned above the `thread_tick()` function, which being called from `timer_interrupt()` on each timer interrupt handling, calls periodically (when the current thread time slice expires — see the `TIME_SLICE` constant) `thread_yield()`. However, we look carefully at `thread_tick()` function we could note that is not the `thread_yield()` which is called, but another function, `intr_yield_on_return()`. Looking at this function we note it simply set a flag, which after the timer interrupt is handled (so after `thread_tick()` and `timer_interrupt()` return) is calling the `thread_yield()`. This is because we want to avoid switching the CPU to another thread (execution), while handling an interrupt in the context of the current thread. We instead on such details because we must note that `thread_block()` must also be called from the `alarm_clock_check()` function, when an alarm clock time expires and the thread waiting for that alarm clock must be awaken (i.e. unblocked). In that case, having the `thread_unblock()` directly calling `thread_yield()`, if the unblocked thread has a higher priority than the current one (as described above regarding the `thread_unblock()` functionality) would not comply the requirement to not call `thread_yield()` from an interrupt routine. This led us to a change of `thread_unblock()` function like this:

```
Require: unblocked_thread, ready_list, current_thread
```

```
Ensure: Preempt the running thread if smaller than the unblocked one
  $new\_prio$ = THREAD_GET_PRIORITY(unblocked_thread)
  $crt\_prio$ = THREAD_GET_PRIORITY(current_thread)
  if $new\_prio > crt\_prio$ then
      if INTR_CONTEXT then
          intr_yield_on_return()                ▷ Set preemption flag
      else
          thread_yield()    ▷ Directly preempt the currently running thread
      end if
  elseLIST_INSERT_ORDERED(ready_list, unblocked_thread)
  end if
```

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 `thread_set_priority()`, 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 the CPU. We identified two situations:

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

   (a) if the new priority is still 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 `thread_yield()` 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 `thread_create()`) or when the thread itself changes its priority (in function `thread_set_priority()`). Correspondingly, we assign values to the real priority field `real_priority` in function `init_thread()` (called by `thread_create`) and in function `thread_set_priority()`. 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 locks. The main idea is that while holding a lock, 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 lock. Being already acquired, the higher-priority thread must wait, being blocked (so, ceasing the CPU) and giving the lock 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 (lock holder) and, indirectly, the higher thread waiting for the lock. 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 lock 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 lock holder has the lock. When the lock 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. `lock_acquire()`, where priority donation could happen, and

2. `lock_release()`, where priority restoration must happen, if needed.

Locks are currently implemented using semaphores (this is because a lock is a particular case of a semaphore). So, the `lock_acquire()`,basically call `sema_down()`, while `lock_release()` calls `sema_up`. Yet, we need to control precisely what happens with thread priorities when a lock is not available and when a lock is released. This is why we decided to implement the locks without semaphores. Basically, their implementation is a sort of copy-paste of semaphore's implementation, and is illustrated below.

Function `lock_acquire()` would look like:

```
void lock_acquire (struct lock *lock)
{
  enum intr_level old_level;

  ASSERT (lock != NULL);
  ASSERT (!intr_context ());

  old_level = intr_disable ();
  while (lock->value == 0)
    {
      list_push_back (&lock->waiters, &thread_current ()->elem);
      thread_block ();
    }
  lock->value = 0;
  lock->holder = thread_current();
  intr_set_level (old_level);
}
```

Function `lock_release()` would look like:

```
void lock_release (struct lock *lock)
{
  enum intr_level old_level;

  ASSERT (lock != NULL);

  old_level = intr_disable ();

  lock->holder = NULL;

  lock->value = 1;

  if (!list_empty (&lock->waiters))
    thread_unblock (list_entry (list_pop_front (&lock->waiters),
                                struct thread, elem));
  intr_set_level (old_level);
}
```

Correspondingly, we must add the similar fields in the `struct lock` structure.

```
    unsigned value;                /* Current value. */
    struct list waiters;           /* List of waiting threads. */
```

The added fields must be initialized in function `lock_init()`:

```
lock->value = 1;
```

```
list_init(&lock->waiters);
```

Regarding the priority donation problem, we should do the following in `lock_acquire()`, on the execution path corresponding to the lock being held, when the current thread must be blocked:

```
while (lock->value == 0)
{
    crt_th_prio = thread_get_priority(thread_current());
    lock_holder_prio = thread_get_priority(lock->holder);

    if (crt_th_prio > lock_holder_prio)
    {
       // priority donation
       lock->holder->priority = lock_holder_prio;
    }
}
```

Similarly, in `lock_release()` we must restore the priority of lock holder to its real value:

```
if (lock->holder->priority != lock->holder->real_priority)
{
    lock->holder->priority = lock->holder->real_priority;
}
```

In practice it is not so simple, though, because a thread could hold simultaneously multiple locks. In that case it could be donated multiple priorities (so, its effective priority being boosted multiple times), due to the multiple locks 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 locks, 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 locks 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 lock 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 locks it holds. Noting that each lock is associated a waiting list, i.e. a list of all thread waiting for that lock, it is very simple to see which such threads have a greater priority than the lock holder, and consider such a priority as a donation. This is the reason we defined the field `acquired_locks_list` for each thread. As with any variable, let us see how it is initialized and used:

- `acquired_locks_list` initialization should be done at thread creation and we chose to do it in `init_thread()`;
- inserting an element (i.e. a lock) in that list must be done when a lock is acquired by the thread, i.e in function `lock_acquire()` on the execution path corresponding the the lock being acquired (after the while loop, waiting for the lock to become available);
- removing an element (i.e. a lock) from that list must be done when a lock is no longer held by a thread, so in function `lock_release()`.

Because we must add the `struct lock` structures to the `acquired_locks_list`, we need to add a `struct list_elem` field in that structures, which we name `acquired_lock_list_elem`.

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 locks. So, the very simple algorithm described above in function `mutex_release()` was replaced by another one that we will place in function `thread_recompute_priority()`, which basically performs the following steps:

1. initialize a current maximum value to the thread's real priority;
2. iterate the `acquired_locks_list` of the thread;
3. for each lock, 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 `thread_get_priority()` function to correctly get the thread's effective priority.

Another problem described in relation to priority inversion is the case a thread holding a lock is waited by a higher priority thread (which, as established, donates its priority to the lock holder), but the waiting thread is in its turn the holder of another lock, 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 lock), if that thread is not waiting in its turn after another thread (holding another lock, in its turn). In order to manage this, we decided to add a new field `waited_lock` in the `struct thread` structure, to keep a pointer to a lock that is waited for by the thread. In case there is no such waited lock, the pointer would be NULL. We must note that there is no need, in this case for a list of waited locks, while, the thread being a single, sequential execution, there is no way to wait for multiple locks in the same time (i.e. while being block waiting for a lock, no execution of that thread take place, such that to be able to attempt taking another lock).

The initialization to NULL of the field `waited_lock` will be done in `init_thread()`, while a valid value (i.e. a pointer to a real lock) will be done when a thread is going to be blocked waiting for an already acquired lock, which is in function `lock_acquire()` just before calling `thread_block()`.

The usage of the `waited_lock` must be done in `lock_acquire()` also, when a thread is going to be blocked, but in relation to the priority donation for the lock holder. We want to place the priority donation algorithm in a function called `thread_donate_priority()`, which will perform the following steps:

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

```
donor = thread_current();
if (thread_get_priority(donor) > thread_get_priority(lock->holder))
{
  lock->holder->priority = thread_get_priority(donor);
}
```

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

   ```
   lock->holder->waited_lock->holder->priority = lock->holder->prior
   ```

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

The last problem we have to deal with in relation to priority inversion is about a thread changing its own real priority, in function `thread_set_priority()`. 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 `thread_recompute_priority()`.

Advanced Scheduler: Multi-Level Feedback Queue Scheduler (MLFQ))

To be determined by students (for teams of four members). Read the given documentation, while all the needed details are described there.

3.2.4 Explanation of Your Design Decisions

Alarm Clock

To be determined by students, if they have other alternatives or note some inconsistencies in the given design.

Priority Scheduler. Fixed-Priority Scheduler

To be determined by students, if they have other alternatives or note some inconsistencies in the given design.

Priority Scheduler. Priority Donation

To be determined by students, if they have other alternatives or note some inconsistencies in the given design.

Advanced Scheduler: Multi-Level Feedback Queue Scheduler (MLFQ))

To be determined by students, if they have other alternatives or note some inconsistencies in the given design.

3.3 Tests

Your are given in your *Pintos* 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).

Alarm Clock

1. alarm-single: this test creates specific number of threads for 1 iteration and checks if the 'timer_sleep' function runs properly (counts for a specific time period;
2. alarm-multiple: this test creates specific number of threads for multiple iterations and checks if the 'timer_sleep' function runs properly (counts for a specific time period);
3. alarm-simultaneous: checks if the order in which the threads 'woke up' is

the same as the one in which they 'went to sleep';
4. alarm-priority: checks that when the alarm clock wakes up threads, the higher-priority threads run first;
5. alarm-zero: tests timer_sleep(0), which should return immediately. This means that the thread is not sleeping;
6. alarm-negative: tests timer_sleep(-100). If there is negative number of ticks, it's supposed to work, not crash.

Priority Scheduler. Fixed-Priority Scheduler

1. priority-change: this tests the case in which a thread with a high priority after lowering his priority will yield and give his right on CPU to another thread;

2. priority-fifo: creates several threads all at the same priority and ensures that they consistently run in the same round-robin order;
3. priority-preempt: ensures that a high-priority thread really preempts (that the highest-priority thread is the first one);
4. priority-sema: tests that the highest-priority thread waiting on a semaphore is the first to wake up;
5. priority-condvar: tests that cond_signal() wakes up the highest-priority thread waiting in cond_wait().

Priority Scheduler. Priority Donation

1. priority-donate-one: this tests the priority inversion problem and checks if after the main thread release the lock the other two threads will acquire the lock in priority order;
2. priority-donate-multiple: this tests the case in which a thread holds simultaneously multiple locks and a new thread with a high priority tries to acquire the lock;
3. priority-donate-multiple2: this is almost the same as the previous one. The only difference is that the main thread releases the locks in a different order;

4. priority-donate-nest: tests the case in which if a thread H is waiting on a lock that another thread M holds and M is waiting on a lock that the L thread holds, then both M and L should be boosted to H's priority;
5. priority-donate-sema: if a low-priority thread acquires a semaphore and other threads with higher priorities are waiting on the same semaphore;
6. priority-donate-lower: this tests the case in which after a thread is donating his priority to another thread which hold the lock and this tread releases the lock, this thread can't recompute his priority until the other threads do not release the lock;
7. priority-donate-chain: this test creates 7 threads and 8 locks, all of which depend on the others, and the purpose is that the main thread should obtain the donation of priority then all threads exit one by one.

Advanced Scheduler: Multi-Level Feedback Queue Scheduler (MLFQ))

To be determined by students.

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.

3.5 Questions that you could be asked

This section must be removed. This is only to give you some hints for your design.

Some questions you have to answer (inspired from the original Pintos design templates), but these are not the only possible questions and we insist that your design should not be based exclusively to answering such questions:

1. alarm clock

    - Briefly describe what happens in a call to *timer_ sleep()*, including the effects of the timer interrupt handler.

- What steps are taken to minimize the amount of time spent in the timer interrupt handler?
- How are race conditions avoided when multiple threads call *timer_sleep()* simultaneously?
- How are race conditions avoided when a timer interrupt occurs during a call to *timer_sleep()*?

2. priority scheduler

   - How do you ensure that the highest priority thread waiting for a lock, semaphore, or condition variable wakes up first?
   - Describe the sequence of events when a call to *lock_acquire()* causes a priority donation. How is nested donation handled?
   - Describe the sequence of events when *lock_release()* is called on a lock that a higher-priority thread is waiting for.
   - Describe a potential race in *thread_set_priority()* and explain how your implementation avoids it. Can you use a lock to avoid this race?

3. advanced scheduler (MLFQ)

   - Suppose threads A, B, and C have nice values 0, 1, and 2. Each has a *recent_cpu* value of 0. Fill in the Table 3.1 (note: you can use `http://www.tablesgenerator.com/` to easily generate Latex tables) showing the scheduling decision and the priority and *recent_cpu* values for each thread after each given number of timer ticks:
   - Did any ambiguities in the scheduler specification make values in the table uncertain? If so, what rule did you use to resolve them? Does this match the behavior of your scheduler?
   - How is the way you divided the cost of scheduling between code inside and outside interrupt context likely to affect performance?

Tabela 3.1: MLFQ Tracing Example

timer ticks	recent_cpu			priority			thread to run
	A	B	C	A	B	C	
0							
4							
8							
12							
16							
20							
24							
28							
32							
36							

Capitolul 4

Design of Module *Userprog*

4.1 Assignment Requirements

4.1.1 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.

4.1.2 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 major requirements of the "Userprog" assignment, described in the original Pintos documentation, are the following:

- *System Calls for Process Management.* You have to implement the system calls *exec()*, *exit()* and *wait()* to create a new process, terminate an existing (calling) process and wait for termination of a child process, respectively.

- *Program Argument Passing.* You have to change new process creation, such that the program arguments to be passed on its user-space stack.

- *Denying Writes to Executables.* You have to write-protect an executable file, while at least one process instantiated from that file is running.
- *System Calls for File System Access.* You have to implement system calls for creating new files (*create()*), accessing data in a file (*open()*, *read()*, *write()*, *seek()*, *close()*).

Some additional (and optional) requirements of the "Userprog" assignment, specific to UTCN / CS OSD course could be:

- *Multi-threading Support for User Applications.* You have to add in kernel multi-threading support for creating and managing multiple threads in user applications. You also have to add corresponding system calls, like: `thread_create()`, `thread_join()`, `thread_exit()`.
- *IPC mechanisms.* You have to add in-kernel support for IPC mechanisms (pipes, shared memory) and the corresponding system calls (also including synchronization mechanisms) for user applications.
- *Dynamic Memory Allocation Support.* Add in kernel support for mapping new areas in the application's virtual address space and also support managing dynamically allocated memory and corresponding system calls.
- *Code Sharing Support.* Add support for sharing common code of different processes.

The way to allocate requirements on member teams.

- 3-members teams
    1. argument passing + validation of system call arguments (pointers) + denying writes to executables
    2. system calls for process management
    3. system calls for file system access

- 4-members teams (exceptional cases)
    1. argument passing + denying writes to executables
    2. system calls for process management
    3. system calls for file system access
    4. support for multi-threading and corresponding system calls
- optional subjects (for extra points)
    - code memory sharing support
    - dynamic memory allocation support
    - IPC mechanisms (pipes, shared memory, synchronization mechanisms)

4.1.3 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.

4.2 Design Description

4.2.1 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".

4.2.2 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.

4.2.3 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.

4.2.4 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.

4.3 Tests

Your are given in your *Pintos* 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).

4.4 Observations

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

4.5 Questions that you could be asked

This section must be removed. This is only to give you some hints for your design.

Some questions you have to answer (inspired from the original Pintos design templates), but these are not the only possible questions and we insist that your design should not be based exclusively to answering such questions:

1. argument passing

    - Briefly describe how you implemented argument parsing. How do you arrange for the elements of argv[] to be in the right order? How do you avoid overflowing the stack page?
    - Why does Pintos implement *strtok_r()* but not *strtok()*?
    - In Pintos, the kernel separates commands into a executable name and arguments. In Unix-like systems, the shell does this separation. Identify at least two advantages of the Unix approach.

2. system calls

    - Describe how file descriptors are associated with open files. Are file descriptors unique within the entire OS or just within a single process?

- Describe your code for reading and writing user data from the kernel.
- Suppose a system call causes a full page (4,096 bytes) of data to be copied from user space into the kernel. What is the least and the greatest possible number of inspections of the page table (e.g. calls to *pagedir_get_page()*) that might result? What about for a system call that only copies 2 bytes of data? Is there room for improvement in these numbers, and how much?
- Briefly describe your implementation of the "wait" system call and how it interacts with process termination.
- Any access to user program memory at a user-specified address can fail due to a bad pointer value. Such accesses must cause the process to be terminated. System calls are fraught with such accesses, e.g. a "write" system call requires reading the system call number from the user stack, then each of the call's three arguments, then an arbitrary amount of user memory, and any of these can fail at any point. This poses a design and error-handling problem: how do you best avoid obscuring the primary function of code in a morass of error-handling? Furthermore, when an error is detected, how do you ensure that all temporarily allocated resources (locks, buffers, etc.) are freed? In a few paragraphs, describe the strategy or strategies you adopted for managing these issues. Give an example.
- The "exec" system call returns -1 if loading the new executable fails, so it cannot return before the new executable has completed loading. How does your code ensure this? How is the load success/failure status passed back to the thread that calls "exec"?
- Consider parent process P with child process C. How do you ensure proper synchronization and avoid race conditions when P calls wait(C) before C exits? After C exits? How do you ensure that all resources are freed in each case? How about when P terminates without waiting, before C exits? After C exits? Are there any

special cases?

Capitolul 5

Design of Module *virtualmemory*

5.1 Assignment Requirements

5.1.1 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.

5.1.2 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 "Virtual Memory" assignment are the following:

- *Paging.* You have to implement lazy loading of process pages, a page replacement algorithm (i.e. second-chance) and swapping.
- *Stack Growth.* You have to add support for dynamically allocated stack.
- *Memory-Mapped Files.* You have to add support for mapping and unmapping files in a process virtual address space, based on lazy loading and swapping, when needed.

The way to allocate requirements on member teams.

- 3-members teams
    1. paging: lazy loading
    2. paging: LRU + swapping
    3. memory-mapped files
- 4-members teams (exceptional cases)
    1. paging: lazy loading
    2. paging: LRU + swapping
    3. memory-mapped files
    4. stack growth

5.1.3 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.

5.2 Design Description

5.2.1 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".

5.2.2 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.

5.2.3 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.

5.2.4 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.

5.3 Tests

Your are given in your *Pintos* 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).

5.4 Observations

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

5.5 Questions that you could be asked

This section must be removed. This is only to give you some hints for your design.

Some questions you have to answer (inspired from the original Pintos design templates), but these are not the only possible questions and we insist that your design should not be based exclusively to answering such questions:

1. page table management

    - In a few paragraphs, describe your code for locating the frame, if any, that contains the data of a given page.
    - How does your code coordinate accessed and dirty bits between kernel and user virtual addresses that alias a single frame, or alternatively how do you avoid the issue?
    - When two user processes both need a new frame at the same time, how are races avoided?

2. page replacement and swapping

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

sign falls along this continuum and why you chose to design it this way.

3. memory-mapped files

    - Describe how memory mapped files integrate into your virtual memory subsystem. Explain how the page fault and eviction processes differ between swap pages and other pages.
    - Explain how you determine whether a new file mapping overlaps any existing segment.
    - Mappings created with “mmap” have similar semantics to those of data demand-paged from executables, except that “mmap” mappings are written back to their original files, not to swap. This implies that much of their implementation can be shared. Explain why your implementation either does or does not share much of the code for the two situations.

Capitolul 6

Design of Module *File System*

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 “File System” assignment are given in the *Pintos* original documentation.

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 *Pintos* 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.

6.5 Questions that you could be asked

This section must be removed. This is only to give you some hints for your design.

Some questions you have to answer (inspired from the original Pintos design templates), but these are not the only possible questions and we insist that your design should not be based exclusively to answering such questions: TBD.