SMP OSP:
a multiprocessor operating system simulator

Martin Bruggink

August 2000

Contents

List of Figures

List of Tables

Introduction

The document you are reading now is the final paper which I wrote as part of my education, which is computer science at the University of Nijmegen. It is the result of my work at SUNY Stony Brook, New York. In it you will find detailed information about the operating system teaching tool OSP (Operating System Project) developed there. OSP is a program used for teaching students who want to learn the concepts of operating systems without having to know all kind of details. A better and more detailed description is given in the next chapter.
The first version of OSP was introduced in 1980. It is written in C and has been used in operating system courses for many years now at many universities all over the world. OSP98 or OSP2[1] is the second version of this tool and is written in Java. OSP2 is still under development, but it is expected to reach its completion soon. In the fall semester this year, OSP2 will be tested during an operating system course at Stony Brook.
I was introduced to OSP during the spring semester of 1999. I started thinking about a subject on which I could graduate from university. Operating systems have always interested me a lot. I had followed the operating systems course about 3 years before and later I was a teaching assistant at the same course. I wanted to do something with operating systems and wanted to do an internship abroad. This is what I said to the teacher of the operating system course, professor Frits Vaandrager. He brought me into contact with professor Scott Smolka who works at the State University of Stony Brook on Long Island, New York. He and professor Michael Kifer, who is also from the University of Stony Brook, are developing OSP2. I was invited to come to Stony Brook and work on the project. In total I worked for 4 months on the project in Stony Brook, also writing part of my final paper, but finishing it in Nijmegen.
Several students have worked and still are working on the OSP project. Most students work part time on the project for about a year as part of there master education. When I was in Stony Brook, 4 students (including me), worked on the project, but there had been several students working on it before. Every two weeks there is a meeting between the students and the both professors. Professors Smolka and Kifer manage the whole project and help the students to make decisions while the students build the implementation.
During these 4 months in Stony Brook I did several things on the project. In order to get to know the program better and to get a better understanding of how it is constructed, I first added a module to the program, which is the ports module. After adding the ports module to the program I was able to put a design of OSP2 on paper. Such a design was never made, at least, it was not put on paper, but there were documents about most of the modules in OSP

[1]In the rest of this paper OSP2 will be referred to as OSP.

though. While writing this design I made suggestions on how to improve the system and found some bugs. Some of these suggestions I implemented and I fixed some of the bugs. The suggestions are discussed in this paper.
The design I made can be useful to students who will work on the project during one or more semesters and who need to understand how OSP is composed. But for myself, the goal of writing down the design not only was getting to fully understand the program, but it was also a practice for the design which I made afterwards.
This second design is called the SMP OSP design. SMP stands for symmetric multiprocessor, which is a way of dealing with multiple processors in a computer system. Multiprocessor operating systems are more complicated than uniprocessor systems, especially the synchronization within the operating system is difficult. So making a new design can be very interesting to the OSP programmer, as well as to the student for whom the tool is meant to be[2]. The first design will be referred to as current design, the second as SMP design. More about SMP systems is written down in chapter 3.

The goal of my work was two folded. On the one hand I tried to improve the current system, by looking closely at the system and doing suggestions for changing certain parts. On the other hand I wanted to build a new design, for simulating an SMP system.
This paper contains both designs. The structure reflects the order of my work on the project. In chapter 1 you will find information about what the program does and how it is used. The current design is discussed in detail in chapter 2. Chapter 3 reports on SMP systems. Some general theory about SMP systems is given as well as examples from some real operating systems, such as Linux. This theory helps to better understand the choices made in chapter 4. This chapter describes the SMP design in the same kind of way as the current design. First some specifications for SMP OSP will be given, followed by the actual design. In the conclusions (chapter 5) I will look at the results and see if these results meet the goals as stated before writing down the SMP design.
A final remark about SMP OSP: though the design is finished, there is no implementation yet. Hopefully, one or more students from the OSP project will try to build SMP OSP in the near future.

Acknowledgement

I want to thank professor Scott Smolka and professor Michael Kifer for giving me the opportunity to come to New York and work on the OSP project. Also I would like to thank professor Frits Vaandrager, the one who brought me into contact with professor Smolka, and guided me in the whole process, also helping me to improve this paper.
Then I want to thank the people who helped me survive in Stony Brook. Roger and Godlind Johnson, for letting me stay in their room above the garage. Not only me, but also my mother, my sister, my girlfriend, and even some noisy friends.

[2]Notice the difference: the OSP programmer, who works on the project and the student who will have to work with the program during an operating system course. Both can be students though.

Then Phyllis Dijkhuizen, for giving me shelter the first month in Stony Brook.
Then of course Mariolein and Michael who invited me and some of my friends over for dinner more than once. Especially thanksgiving was great!
David and Allison, good luck in Boston, thanks for the dinners, and I'm sorry I wasn't there at your wedding. Maybe I'll see you some day in Germany.
Oh, and last but not least: my mother, thanks for calling me every week at the proud lab.
To all the people who visited me: "Dat doen we dus nooit weer!".

Martin Bruggink
University of Nijmegen
martinb@sci.kun.nl

1 An overview of OSP

In this chapter, before dealing with the details of the OSP design, first some information is given about tools like OSP, allowing to compare OSP with other operating system tools. After this discussion, the way to use OSP will be described, giving more information about how OSP works and what possibilities the student has in working with the program.
Then a more or less global view of the OSP design will be given, also containing some details of the system. This way the reader will have better understanding of the way OSP works and what kind of operating system is implemented. In the next chapter all modules from OSP will be discussed in more detail.

1.1 Operating system tools

In order to teach first or second year students the concepts of operating systems, some programs have been developed at several universities. The best known are probably Minix, Nachos and OSP. All these programs implement a simplified operating system, which the student has to extend or change. It is tried to avoid the student from having to know all kind of details by keeping the operating system simple. This way the student can get familiar with the main concepts in operating systems.
OSP is intended to be used in introductory courses in operating system design. OSP simulates both the hardware and the software in the system, while the student can implement part of the operating system. Because the hardware is simulated, the student does not have to know very specific information about the machine the operating system is meant to run on. To put in the words of the makers, 'the OSP tool exposes students to many essential features of operating systems, while at the same time isolating them from low-level machine-dependent concerns'[1].
In comparing the operating system tools mentioned above there are certain characteristics about the tools one wants to find out. The following list with five characteristics will be discussed.

1. Realism
2. Simplicity
3. Possibilities for optimization
4. Feedback and user friendliness
5. Kind of assignments

An important question to ask is how realistic does the system seem to be. Todays operating systems are complicated pieces of software. It would be too

much to ask a first year student to implement even a part of such a system. That is of course the reason why these operating system tools have been developed. But by creating a very simplified operating system, the student might get the wrong ideas about the functionality of an operating system or the functionality of the hardware, which might me simulated as well (as in OSP and Nachos). These kind of false impressions should be avoided. As the makers of Nachos put it the introduction to their tool:'In building Nachos, we were continually faced with tradeoff between simplicity and realism'[2]. In all operating system tools a balance has been tried to find between simplicity and realism, which are the first two characteristics mentioned here.
A third characteristic is what possibilities does the student have for optimizing the system code and what will be the results in the execution of the system. Just like in real operating systems it is important to make optimizations. This is one of the main goals in operating systems research nowadays. So the questions to ask here are what are the possibilities for improving the system and how can these improvements be measured. A student may not be motivated to optimize the system, when he or she is not able to see if these improvements really work. Not only the student will be interested in this kind of feedback, it can also be useful to the teacher when grading the work of the student.
The fourth characteristic from this list is the difficulty for the student to work with the program. Does the student have to know a lot of details before starting to code, or does he or she only have to be informed about the global design? And what kind of feedback is given back to the student? When a code error has been made, will the system return a message giving information about what went wrong or just stay in some form of deadlock? A lot of time can be spent on fixing bugs, this time is better spend by learning more about the concepts of operating systems.
The final characteristic dealt with here is about assignments. These tools are meant to be used for teaching, so assignments are very useful. All three tools are provided with assignments, but some tools are build in such a modular way, that several topics can be clearly distinguished, each topic related to one major assignments. Also important is how these assignments are linked. Can a student skip an assignment without getting behind on schedule too much, or can an assignment not be made before finishing the one before, like when building a tower?
This list only contains five characteristics and of course many others can be given. An interesting question to the teacher can be, how are the manuals provided with the tool. What are the costs for buying the manuals (all programs are free)? This list focuses on characteristics from a tool itself, not on things that come with the tool, like a manual. It probably gives the most important characteristics of an operating system tool itself to take into account when choosing one of the tools in preparing a operating system course.
To see how OSP fits in between the other operating system tools, for each of the three tools a short description will be given, especially referring to the above list. In [4] the reader can find some more information about these tools.

Minix

Minix, developed by Tanenbaum and Woodhull [3] and the basis for Linux, is a complete operating system that runs on several platforms, for example on Intel's

x86 series. There are also some simulation tools available, which make it possible to run Minix as a single UNIX process. Because Minix is a complete operating system, it is also the most realistic of the three. This has a disadvantage though: it is not that simple for a student learn to understand how the operating system works. Minix is written in C, but also contains some machine code.
The advantage here is that the student knows that he or she is working with a real operating system. Understanding such a program and changing it in a proper way can give insight as well as a certain kick. A C-compiler has been built, which runs on Minix, allowing the student to write his or her own programs and run them in Minix.
The developers have tried to keep the system simple, only allowing the necessary things in the code. But this could not prevent them from sometimes having to write code on a very low level of abstraction.
Looking at possibilities of optimization, there are plenty of course. The student can write user programs to run in Minix, testing the changes made in the operating system. This can take some time since the system does not give any more feedback than a normal system would do. A code error in the operating system can result in not starting up Minix, giving little information about what went wrong.
The last characteristic from the list is the kind of assignments that are possible with the system. A lot of assignments can be thought of in Minix. There are several topics to choose from, like memory management and the file system. The student will probably first have to read a lot of information about the part of the operating system he or she has to deal with.
Concluding, Minix can be very interesting to students, especially the curious ones who are willing to spend a lot of time in the operating system course. With good guidance of teachers there should not be too much trouble in finishing an assignment though it may take some time. There is also a book discussing all the code available[3]. The reality of Minix can be very interesting to the student but he or she probably will not have the time to deal with a lot of topics.

Nachos

Nachos is less realistic than Minix because it runs as a single UNIX process, rather then directly on the hardware. Figure 1.1 shows how each of the tools will be executed. Nachos simulates a standard instruction set (MIPS R2/3000) and allows the student to write programs in this set. The Nachos simulator executes each instruction sequentially, one at a time.
In Nachos, the student is provided with a working example of a simple operating system, leaving a lot of space for improvements. The student has to add functionality and improve the performance of the system. Micro-benchmarks are available to test the improvements.
There are 5 major assignments defined in Nachos, each of them is designed to build upon previous ones. The reason for this kind of dependance is that 'this reflects part of the charm of developing operating systems: you get to "use what you build"' as quoted from the developers[2].
Because the operating system is very simple at the beginning, the student will not have to know a lot of details when starting with the first assignment. The operating system is written in a subset of C++ and a manual is available. So it

seems that the student has to spend less time reading about the system before starting to work with it him or herself.

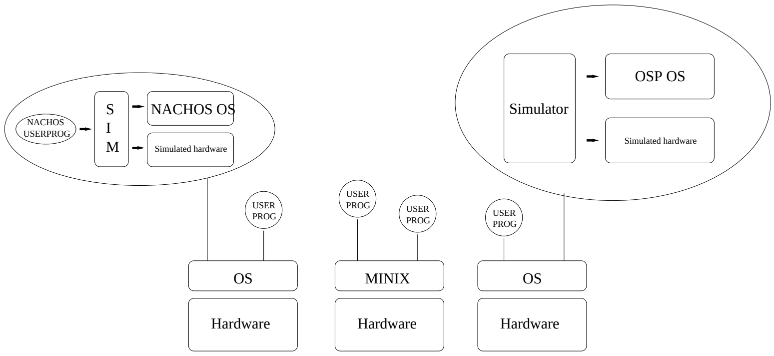

Figure 1.1: *Relation of operating system teaching tools with the hardware the systems run on.*

OSP

OSP is the least realistic of the three, but probably also the simplest one. In OSP, the hardware is simulated, as well as the software (unlike NACHOS, where the software is not simulated). The operating system is written in Java and the compiled version is already available from the beginning, though the code is not. The student has to implement certain system calls and interrupts, which are clearly separated in several modules. The student can link his or her part of the system to the rest of the system and execute the simulation.
The system does not give much feedback about the system performance though it does give feedback about mistakes in the students code. If an error occurs, OSP will return a message containing information about the error.
OSP is probably the easiest one to start with for a student and chances are high the student will deal with several topics within a single operating system course. The system is less realistic than the other two, optimizations are less interesting since one cannot see how the system reacts in terms of speed. But OSP is the only tool which does error checking. That means, the tools checks to see if the students implementation is correct.

	Minix	Nachos	OSP
Realism	++	+/-	-
Simplicity	-	+	++
Optimization possibilities	++	+	+/-
Feedback	+/-	+/-	++
Assignments	+/-	+	++

Table 1.1: *The three operating system tools compared.*

All the tools deal with the general topics within operating systems, like memory management and file system. Nachos is the only tool which also deals with

networks. This is not yet available in OSP2 though it was in the first version of OSP. A network module might be added later.
Each tool has its pros and cons. Minix seems to be the most realistic, OSP the least realistic. But OSP is much easier to use by leaving out a lot of details, which is not the case with Minix, Nachos seems to be somewhere in between. It is up to the teacher to find out what is the best solution for his or her class.

1.2 Using OSP

In this section the usage of OSP will be discussed. Some information about the assignments is given as well as how students will have to deal with them.

1.2.1 Assignment

In OSP2, currently six major operating system modules are defined, which can be implemented by the student. These are Memory, FileSys, Devices, Tasks, Threads and Ports. Each assignment is based on a single module and asks for the implementation of one or more system calls or interrupt handlers from a certain module. Figure 2.4 shows which methods have to be implemented for each of the modules. The functionality of each of the modules will become clear in chapter 2.
The usage of OSP is shown in the next example. Suppose the teacher wants the students to do the assignment from the Threads module. The teacher compiles OSP, leaving out the Threads module. All six modules are already implemented by the OSP programmers, but the student is not supposed to see this code. He or she will receive the compiled version of OSP without the Threads module, as well as one or more header files (one for each class defined) from the Threads module. Such a header file contains the definition of a class and its methods used in this module, forming the interface with the rest of the system. The student has to complete these header files by implementing the methods defined in it.
In the case of the Threads module, the student has to implement the following methods: create(), kill(), suspend(), resume(), dispatch()[1]. Interfaces with the other modules of the operating system are also provided, so the student can call methods which are defined in the other modules.
When the student has finished coding, the compiled version of OSP can be linked to the header file, which the student has completed. This results in a complete version of OSP, ready to be executed.
The teacher and student do not have to find out how to do the compiling and linking. A generator is made for the teacher which helps him or her to create the assignment. The teacher has to specify which module(s) should not be compiled, the generator takes care of the rest. For the student, the generator will create a make file. Running this make file takes care of linking the students part to the rest of the system.
The student can not start coding without knowing something about the rest of system. A manual[2] can be used by the student, which describes all the methods from the operating system, as well as from the simulated hardware. For each

[1]In most cases methods are printed without parameters, making it easier to read.
[2]This manual is not yet available for OSP2.

assignment some information about how to deal with the problem is given in the manual.

1.2.2 Simulation

After the coding is done, the simulator can be started by executing the OSP program the student has compiled. The generator not only compiles the six modules, but also the hardware and the simulator. Details about the hardware and the simulator can be found in the chapter 2.
A GUI (Graphical User Interface) helps the students to set certain parameters which influence the simulation. The student can for example increase the number of simulated task creations and killings. Other things that can be influenced are the length of the simulation, what kind of messages should be generated to the log file, but also how many I/O devices should be simulated in the hardware and many more. By changing these parameters, the student can test his or her part of the system. To return to the example in 1.2.1, the number of threads being created and killed can be increased. This way especially the Thread module will be accessed during the simulation. The GUI saves these settings to a parameter file. An example of such a parameter file is given in appendix A.
After setting the parameters the actual simulation can be started. Software is simulated in OSP by issuing system calls to the operating system. Also interrupts from the hardware are generated. Which system calls are simulated and how many depends on the parameters from the parameter file.
OSP checks the correctness of the students code by looking at the program's state before and after calling the students method. It keeps a copy of the system, comparing the state of the copy with the state of the actual operating system, parts of which are implemented by the student. If the states of the operating system and the copy seem different after the execution of the students method, the program stops executing. An error message will be produced, which contains specific feedback about what went wrong.
During the simulation, feedback is given about which system calls and interrupts have been generated and how many. Also messages about the simulation are printed to the log file. The student can also print messages to this log file from his or her part of the system, methods are provided. Another way to get feedback from the simulator is by using snapshots. During simulation a number of snapshots are made of the system (the number is defined in the parameter file), giving information about the state of the system. Some details from such a snapshot are the number of living threads, which are asleep and which are ready to run, which parts in memory are used and by which task are they occupied. For each module specific information is given. This way the student can see what happens during the simulation and what her or she should change to improve the system.

1.3 Global view and design issues

In the previous section OSP was discussed from the user perspective. In this section the design will be discussed without going into too much detail. First some information about all modules forming OSP and their relations will be given, providing the reader with a global view on OSP. Then the way in which

OSP gives feedback to the student will be discussed. Finally, some more information about the operating system itself and the hardware will be given. In section 2 each module will be discussed in more detail.
A interesting difference between a program such as OSP and other software, is that the code of OSP is always changing, even after releasing it. Normally, software is delivered to the client and the client will not change the program. In OSP, the programmer has to check code within its own program, because the students adds pieces of software.

1.3.1 Packages

OSP2 consists of a number of packages, as defined in the Java files. A short description of each of these packages will be given.

- **EventEngine**

This package is responsible for simulating the software and generating interrupts in the hardware. The event engine makes sure the simulation stays running and keeps track of time. The event engine will be discussed in section 2.2.

- **Hardware**

The hardware package is responsible for simulating the hardware. The student will not have to change this package, but he or she will have to know how to access the hardware. For example, he or she must be able to remove a process from the processor. The hardware consists of a processor, a memory management unit, and several I/O devices. More about the hardware can be found in section 2.3

- **Interrupts**

This is a small package, which defines the general interrupt handler. It is not a separate assignment because it is too small. All specific interrupt handlers, like the disk interrupt handler, are defined in one of the operating system packages.

- **Memory**
- **FileSys**
- **Devices**
- **Tasks**
- **Threads**
- **Ports**

These packages together with Interrupts form the students part of the operating system. In each of these packages one or more classes are defined. A student assignment always consists of implementing one or more of these classes.
As said before, all these packages are already implemented by the OSP programmers. The project generator makes sure the correct classes are left out in preparing an assignment. The other classes are compiled with the rest of OSP, and should not show any errors when running the simulation. A separate section is devoted to each of this packages further on in this chapter.

- **IFL (Interface Layer)**

The IFL package is responsible for checking the students code on errors. For every class in the six packages listed above there is a class in the IFL package as well.[3] How the error checking is done is made clear below.
The IFL module is also responsible for setting up parameters when system calls are issued by the event engine as well as for collecting information about the state of system when for example a snapshot is taken.

- **GUI**
- **Generator**

The GUI package is responsible for displaying the GUI through which the student can change parameters and receive feedback about the simulation. In the generator package a separate tool is implemented for creating assignments. This is described in section 1.2.1. These two packages will not be discussed any further in this document, OSP can be run without them.

- **Utilities**

Utilities defines some classes which are used by all the other module. Class RndGen can be used to generate random numbers. By using a self defined class for generating such numbers the OSP programmers have made sure that each simulation is exactly the same when the same parameters are used and the students code stays more or less the same. This makes it easier for the student to debug his or her implementation.
Besides RndGen, some classes are defined for using queues which will not be discussed here. Classes SysOut, UsrOut, Out, DbgOut provide methods to print messages to the output screen in the GUI and the log file.
An import class in this package is GlobalVariables. It mainly defines static variables and constants, which are used through the whole program. Almost every class in OSP is a subclass of GlobalVariables. This sometimes makes it difficult to read the code, because the programmer does not explicitly has to state where these variables are stored.

Modules

In the rest of this document packages will be referred to as modules. Modules and packages in OSP are not exactly the same, which will become clear in the next section. Figure 1.2 shows the interaction between the modules. The IFL module, the six major modules from the operating system and the Interrupts module together form the operating system (see section 1.3.2).
All modules use classes defined in Utilities. The operating system talks to the hardware, for example when starting I/O. The hardware talks to the operating system when generating an interrupt. All modules also communicate with the event engine because this module takes care of all simulations. This will become clear later on.

1.3.2 Operating system module

The operating system module is the composition of the IFL module, the six major modules and Interrupts. The six plus Interrupts are also called the students

[3] All names of the classes in the IFL module begin with IFL.

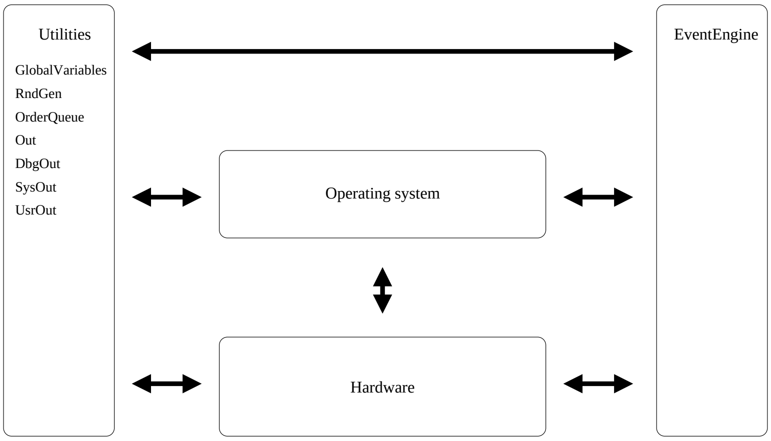

Figure 1.2: *Interaction between the OSP modules. The GUI and Generator modules are left out.*

module. The students module is not defined as a separate package in OSP, it is a way of referring to this part of the operating system. Figure 1.3 depicts this model. The operating system can be defined as the composition of the IFL module and the students module.

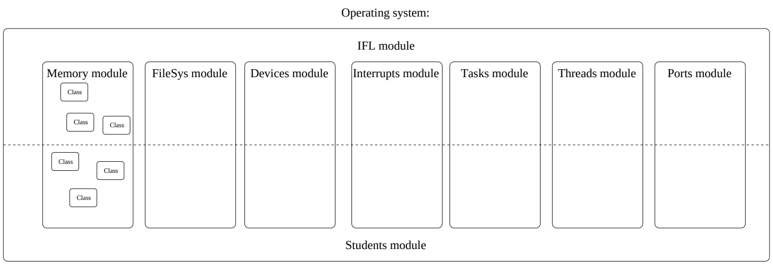

Figure 1.3: *The modules in the operating system*

The 7 modules from the operating system can each be divided in an IFL part and a students part.[4] The IFL part is responsible for doing error checking, keeping its own copy of the operating systems data structures. The students module is the part which is implemented by the student. In the rest of this document, when referring to one of the operating system modules, like Memory, both the IFL part and the student part are meant. The reason that the IFL module is also part of the operating system is that not all functionality of the operating system is implemented in the students module. To do error checking some attributes and functionality are stored in the IFL module. This is explained in the next section.

[4]The interrupts module also has a IFL part and a students part, though no assignment is defined for it.

1.3.3 Error checking

The error checking done in the IFL module is implemented using the inheritance mechanism from object oriented programming. For each class in the students module a class in the IFL module is defined. Each class in the students module inherits the methods defined in the associated class in the IFL module. This is done by defining each class in the students module as a subclass from the associated class in the IFL module. This definition will be put in the header file by the generator tool. The student is not allowed to change the class definition. How this works is shown in the next example. Suppose a certain task has to be killed either by the simulated software or from somewhere in the operating system. Then kill() should be called. This method is defined in class IflTaskCB, which is part of the IFL module. The method is made final so it can not be overridden by a method in the students class.
In the students module a method called do_kill() [5] has to be provided. This method is called from kill() in the IFL module. In do_kill() the student removes the specified task from the system. When the method returns, the IFL module checks to see if the students removed the task properly. For example, it can check if the swap file was closed properly. If not, it returns an error to the GUI and log file, after which the simulation stops.
The next piece of Java code shows how this is implemented.

```
/* From the IFL module: */
public class IflTaskCB extends GlobalVariables
{
    final public static long kill(TaskCB task)
    {
        /* Save current state */
        do_kill(task);
        /*
           Compare current state with previous state
           If current state not as expected
               return error message
           Else
               update IFL state
        */
    }
}

/* From the students module: */
public class TaskCB extends IflTaskCB
{
    public static long do_kill(TaskCB task)
    {
        /* Implemented by the student */
    }
}
```

This example shows how all methods defined in the students module are always

[5] Every method provided by the student which is accessed from the IFL module begins with do_.

accesses via the IFL module. It is not allowed to call a method beginning with do_, thus defined in the students module, directly. Always the associated method defined in the IFL module should be called.
Because the students module is accessed via the IFL module it is possible to give specific feedback to the student about his or her implementation. The IFL module stores a copy of most data structures and updates these structures each time a method is called. After the method from the students module returns, the IFL module can compare its copy with data structures from the actual operating system. This way the IFL modules has complete control over what is happening within the operating system. More about the data structures will be explained in the next section.
As stated before, the IFL module not only does error checking, it also sets up parameters and prints information about the current state of the program to the log file. It serves as a layer between the event engine and the (students part of the) operating system, thus the name interface layer(IFL).
Error checking is one of the main characteristics by which OSP distinguishes itself from the other operating system tools. The OSP programmers try to implement the system in such a way that specific feedback is provided in case something seems wrong with the students code. This way, the student will spend less time in finding out what is going wrong.

1.3.4 Attribute storage

A class in the IFL module has two kind of attributes. Firstly the ones that can be accessed from the students module and secondly the ones that are not accessible from the students module.
The ones that are accessible from the students module are actually part of the operating system but have been moved to the IFL module. This way it is easier to access these attributes from the IFL module. These attributes can be accessed from the students module through the provided access methods in the IFL module. The student does not know how the IFL module is implemented. Only the access methods which can be used to access the attributes in the IFL module, are provided.
The second type of attribute, which is not accessible from the students module, is in most cases a copy of a data structure defined in the students module.[6] This way the state of the actual operating system can be compared to the state of the IFL copy. In other cases these unaccessible IFL attributes are used to determine the parameters of system calls. For example, in IflTaskCB a static variable is defined called iflTaskList. This is a static list holding references to all tasks created by the system. It is used by the simulator to pick a task when a task should be killed.
The reason for moving attributes from the students module to the IFL module and keeping copies of operating system structures is to do error checking. Three ways of doing error checking can be recognized in OSP:

- An attribute is stored in the IFL module. The student can access such an attribute via interface methods. By keeping the attribute in the IFL module it is easy to do error checking, since all that needs to be done is checking this attribute. The disadvantage is that it takes away responsibility from the

[6] Names of attributes not accessible from the students module always begin with IFL.

students module and sometimes the student has less freedom in implementing the students module because he or she can not choose how to implement the attribute.

- The attribute is stored in both the students and IFL module. The IFL module keeps a copy of the attribute. The student can implement the attribute the way he or she wants.
- An extra data structure is added to the class. This data structure keeps track of what happens during the execution of system call. Each time a method is called, the IFL module updates this object. After the system call return, the IFL module can check the attached data structure to see if all things are correct. This method is only used in the memory module, because this a complex part of the operating system.

Depending on how much error checking should be done, one of these methods is chosen. Most of the times it seems best to have the student implement the attribute and keep a copy in the IFL module. But the OSP programmers do not want the student to have to deal with small details, thus only the important data structures are also stored in the students module.

1.3.5 Operating system design

Next, the functionality of the operating system will be described. In short the responsibilities of each of the modules in the operating system will be discussed. What kind of operating system is implemented in OSP? All functionality is stored in the kernel, which is the same as the operating system (IFL plus student module). Figure 1.4 (taken from [4])shows how kernel functionality is executed. A thread can do requests to the kernel by issuing a system call. The processor jumps to kernel mode and the request will be processed by the thread itself. The major advantage here, at least in a real operating system, is that no process switch is needed when doing a system call, which saves time. It is the same thread which executes the code, only a mode switch (from user to kernel mode) is necessary. The advantage in OSP is hardly noticeable, since a process switch is not that difficult. The speed or reaction time of the system is not very important to the OSP programmers.
The only time no user process (thread) is running is in the case of a process switch. At that particular moment, execution takes places outside of all processes [4].

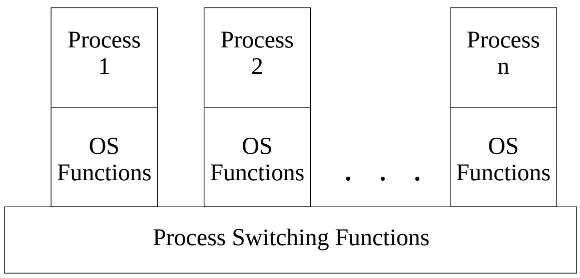

Figure 1.4: *User processes execute kernel code*

Before explaining which data structures are defined in the operating system, first something needs to be explained about the hardware design. The hardware consists of a single processor, one memory management unit (MMU) and several I/O devices, of which one is the swap device. There is no memory simulated, because no actual data is stored in OSP. Only the size of the memory is defined. More about these simplifications can be found below.

Next, the operating system functionality will be described without going into detail. Maybe some question will arise while reading this. Hopefully these questions will answered later on when discussing the modules separately. In OSP, tasks hold all the resources, like threads, ports and a swap file. Threads can be seen as light weight process which execute user programs but operating system code as well. In OSP there is no user code to execute, but a thread can be in several states: running, waiting, blocked and killed. If it is running, the processors is executing the code from the thread. If it is waiting it wants access to the processor. If it is blocked, it is waiting for some event to happen, like some I/O to end. If a thread is in the killed state, the thread was killed and cannot be used anymore.

A task can also have certain ports. A port is used to send messages to other tasks. The messages are empty, the only thing which is defined is the length of each message. Ports can be created and killed during the simulation, just like tasks and threads.

In the hardware, a number of disks are available. For each disk a device is used to control the disk. A device processes I/O request (in the form of I/O request blocks, or short, IORB) and keeps track of how much a disk has been used.

The file system is responsible for mapping files to disks. The file system keeps track of which file or directory is stored on which device and where exactly on that device. These files or directories can be created and removed. A file can be opened and closed, read and written. Symbolic links can be made as well. Again, no data is actually stored in the files.

In the operating system, the memory is defined as a number of frames. Each task has a number of pages, which can be stored in the frames. The memory module is responsible for keeping track of this relation between frames and pages. The memory module also implements swapping. The MMU is responsible for translating virtual memory addresses to physical addresses. Sometimes the MMU interrupts the processor to tell that a certain address is not valid. The processor calls the general interrupt handler which calls the page fault handler from the memory module. The page fault handler must swap the correct page from the swap file into memory.

Simplifications

As already stated before, some simplifications have been made in OSP. These simplification have been implemented to make it easier for students to work with the program, putting the focus on the concepts of the operating system, not on the details. For example, no actual data is stored in memory or moved to or from a disk. There are some more simplifications, some of which will be discussed here, some will become clear in one of the next sections.

Another simplification is that interrupts cannot occur while the processor is in kernel mode. This is a different kind of simplification than the one mentioned before, because one can imagine a system in which no interrupts occur while the processor is in kernel mode. It is mentioned with other simplification because

it does make the system easier to implement.
The last simplification discussed here is the fact that the size of a frame is the same as that of a block in all the disks in the hardware. This makes it a lot easier to move (virtual) data between a disk and memory.

2 OSP current design

In the previous section, a comparison was made between three operating system education tools. It turned out that OSP is the easiest to work with and the only one which gives specific feedback to the student (error checking), but also the least realistic one. This shortage of realism is not only the result of the simplifications, the fact that OSP runs as a single UNIX process or that no user programs run on top of the operating system. It is the result of certain choices made in the design as well. Also certain execution traces occur during simulation which are not realistic. In this chapter, when describing each of the modules separately, suggestions will be given to make OSP more realistic. These are most of the times just small suggestions, sometimes about the design, sometimes about the implementation, but they can make OSP more realistic. Making the system more realistic, without losing the simplifications, will result in a system which is easier for the student to use. Most of these suggestions already have been implemented now.
First, some information about the figures used to describe the design will be given. Then, the event engine, the hardware and the operating system modules will be described.

2.1 Interpreting the design

To understand the design, some knowledge about object oriented design is required, though only a very basic one. In order to explain how all parts of the system are designed, several diagrams are used. The diagrams are made according to the UML specification (see [5]). Three kind of diagrams are used:

- class diagram
- collaboration diagram
- state diagram

Each diagram shows a model of the system from a certain point of view. Not all views are useful in all parts of the system though. Sometimes a state diagram can give much more information than a collaboration diagram. Thus not all diagrams are given for all parts of the system. Next, for each of these types, some information about how to interpret the diagram is given.

2.1.1 Class diagram

A class diagram shows a collection of classes, their attributes and methods, and the relationships between these classes. It helps to visualize the structural or static view of the system. Of the three type of diagrams, class diagram are used the most in OSP.

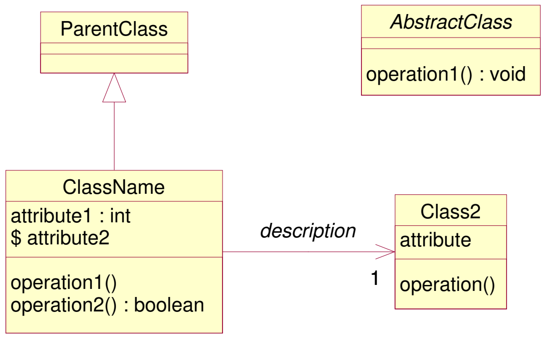

Figure 2.1: *Example of a class diagram*

Figure 2.1 shows an example of a class diagram. Each class is drawn as a rectangle. In the top of the rectangle the class name is printed. Underneath it are the attributes, followed by the operations.
In the figure, class ClassName[1] has one reference to a class called Class2. The description gives a meaning to the relationship. In this case each instance (object) of ClassName, holds exactly one reference to an instance of Class2. The number printed underneath the arrow, specifies the number of references. This can be zero or one(0/1), zero or more (0*), exactly one (1) or one or more (1*). In the example class ClassName is a child of ParentClass (notice the difference between the arrows). This means that the child inherits all attributes and methods (= operations) from the parent. Class2 implements the class called AbstractClass. This means that Class2 implements all methods defined in AbstractClass, in the example only operation1().
Sometimes an attribute is static. This means that the attribute is not defined for each instance of the class, but only once for the class itself. When an attribute is static, a dollar sign ($) is printed in front of the attributes name.
Attributes are of a certain type. This is printed next to the attribute. In the example, attribute1 in ClassName is of type int. The type of attribute2 is not defined. An operation can have a return type defined. This is printed next to the operation. operation2() in ClassName returns a value of type boolean.
Sometimes the module in which the class is defined is printed below the class name. For classes outside the module attributes and methods are not shown.

2.1.2 Collaboration diagram

A collaboration diagram shows how objects interact with each other. It shows what kind of messages are sent between objects. This is useful for showing how the objects of all the classes and modules communicate with each other.
In a collaboration diagram an object is depicted as a rectangle like in the class diagram. Only the class name is shown in the rectangle. Other information

[1] When referring to methods in the text, most of the times, the parameters are left out, like this: method(). Attributes and classes are printed like in the Java code standard, including the capitals: ClassName with severalAttributes. Class names always begin with a capital, methods and attributes start in lower case.

about the class can be found in the class diagram. If an object of a certain class communicates with an object of some other class, a line is drawn between the two rectangles, representing the objects. The kind of message is shown by adding the name of the method which is called, since sending a message to an object means calling one of its methods. The arrow shows in which direction the message is sent. Only the name of the method is shown and parameters are left out, otherwise the diagram would soon become too specific and too complex to read.

2.1.3 State diagram

State diagrams show the states of a certain object. A state is shown as a rectangle, the name of the state printed inside. Only the first state is different, it is a black dot. A note can be attached to a state containing some information about the state in normal text. When an object can jump from a certain state to another, an arrow is shown between these states. So one can see how an object may reach a certain state, beginning the init state.

2.2 Event engine

At the heart of OSP lies the event engine. The event engine is responsible for simulating requests (system calls) done by the software on top of the operating system and simulating interrupts done by the hardware. There are three kind of interrupts: disk interrupt, memory interrupt and timer interrupt. In figure 2.4, the methods with an asterisk are the system calls. They are called from the event engine, but can be called from within the operating system as well. The methods without an asterisk are only called from within the operating system. In OSP, these actions (interrupts and system calls) are called events. Some events are generated more often than others. This depends on the values defined in the parameter file.

In the event engine a class called SimEvent is defined. An object from this class consists of a time stamp and a reference to an object from the event tree (see below). The events (instances of SimEvent will be referenced as events in the rest of this chapter) are stored in the event queue. In this queue the events are ordered on their time stamp, the one with the lowest time stamp at the beginning of the queue. This is the event which will be processed next by the engine thread. The engine thread initiates all the events.

So, the three main parts of the event engine are the engine thread, the event tree and the event queue. The engine threads initiates all the events. The event tree is used by the engine thread to initiate the event. Also, the event tree is used to create new events, which are put on the event queue.

The engine thread is a sub class of the standard Java class Thread (see [6] for more information about programming in Java using threads). Initially, only one thread is running in OSP, the engine thread. Later in the simulation, other threads are created as well. Why will be discussed in section 2.2.6. First, a closer look at the event tree will be taken.

2.2.1 Event tree

Events are put on the event queue with the help of the event tree. The objects in the event tree determine which events are generated as well as the time between each event. This interval can be fixed (useful for generating timer interrupts) or variable, using a certain random value. The type of object determines what kind of interval is chosen. How this works is explained below.

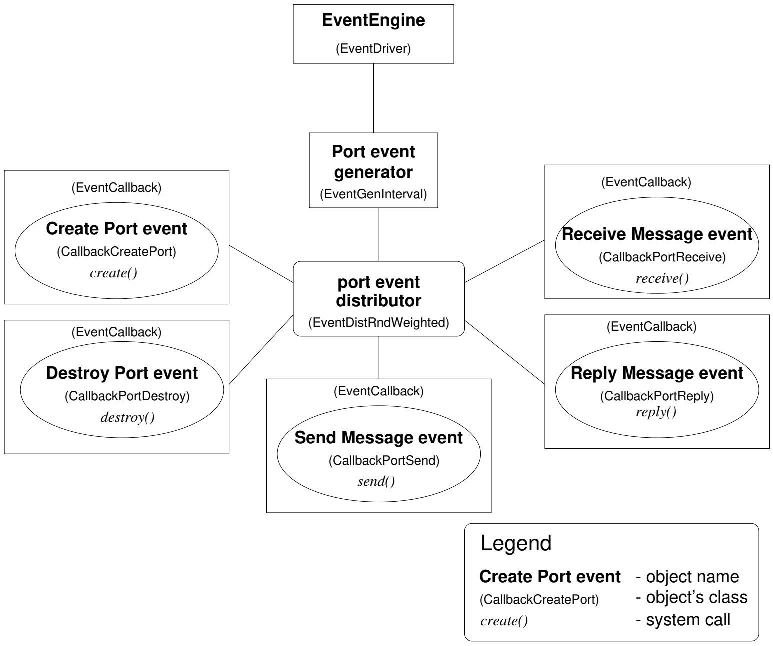

Figure 2.2: *Example of an event tree. Just a part of the whole event tree in the event engine is shown.*

At the beginning of the simulation, the event tree is build up. This is done by calling buildEventtree() defined in each of the operating system modules in the IFL module. Each module will attach a sub tree to the general event tree. Part of the event tree is shown in figure 2.2. At the top lies the event engine. Every object in the event tree holds a list of child objects. The figure only shows one sub tree, the one attached by the ports module. Other sub trees are attached from Memory, Filesys, Threads and Tasks. Devices does not have a sub tree attached, this module does not provide any system calls, only functionality to the rest of the system.

The event tree stays more or less the same during the simulation. Only some new objects are attached to the tree during simulation in order to start interrupt events. This is explained in section 2.2.5.

Each object in the event tree is an instance of a subclass of EventEngObj (see figure2.3 for the class hierarchy). The difference between the objects in the event tree is in the way they handle messages which can be sent down the tree. Two kind of messages can be sent down the event tree, **evaluate** and **activate**. The activate message is used to start the simulation of a certain event and the evaluate is used to put a new event on the event queue. The object association diagram of the event engine shown in appendix 1 shows how the objects in the event engine are associated.

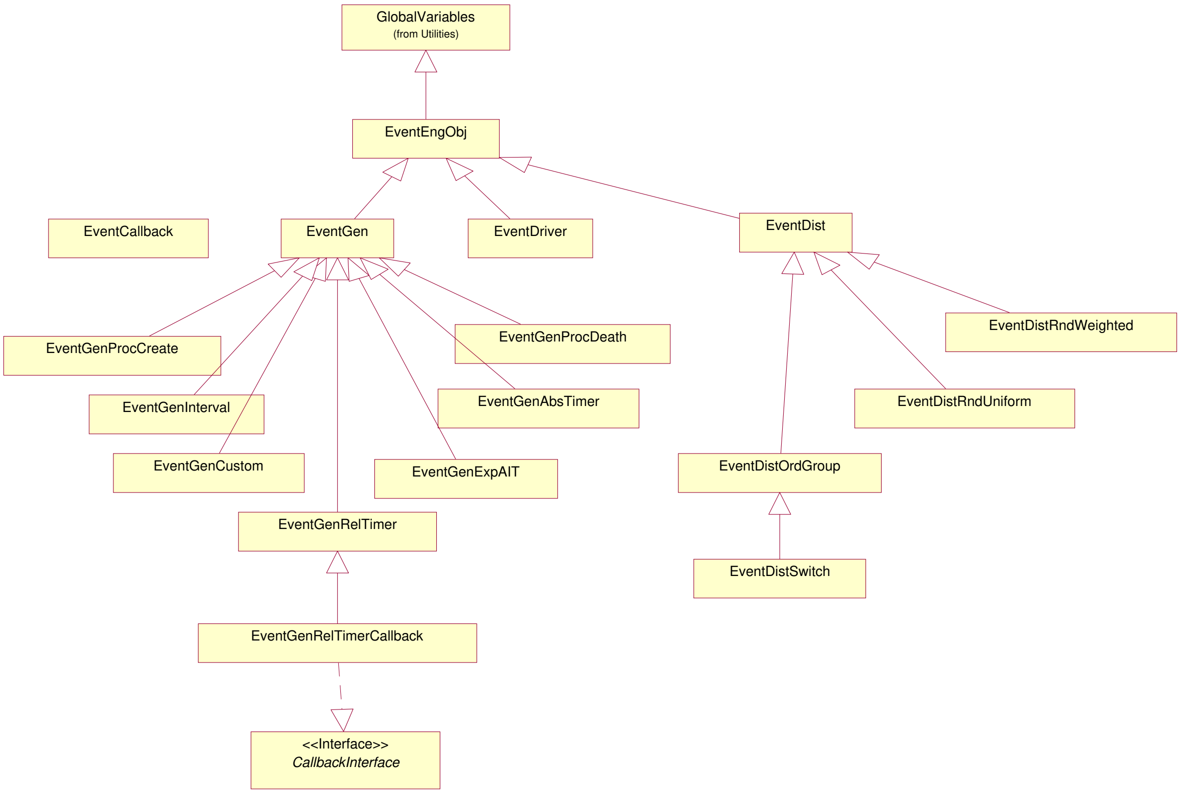

Figure 2.3: *Class tree of the EventEngine module. Methods and attributes are left out.*

Maybe the information given this far about the event engine is a bit confusing. To give a better view of the use of the event engine an example of how one event is simulated will be given next. Assumed is that the event queue already contains some event. How the event queue will be filled at the beginning of the simulation is explained later.
First an event is pulled of from the event queue. This is the event with the lowest time value, thus it should be processed next. The event holds a reference to an object in the event tree. Suppose this is the port event generator as shown in figure 2.2. To start a port event simulation, the activate message is sent to this object, which means that activate() is called, a method defined in each object in the event tree. The port event generator is an instance of the class EventGenInterval which is a subclass of EventGen (see figure 2.3). When an instance of EventGen receives the activate message, it forwards the message to all of its direct child objects. Thus, in this example, the message is only sent to the port event distributor.
The port event distributor is an instance of EventDistRndWeighted, which is

a subclass of EventDist. When an instance of EventDist receives an activate message, it does not forward the message to all its direct child objects in the event tree, but it picks one and then forwards the message to this object (actually it might pick several objects to forward the message to, but in this example only one is chosen).

At the leaves of the event tree, there is always an instance of EventCallback attached. It holds a reference to a callback which implements the callback interface. In figure 2.2 this is depicted as an oval (callback interface implementation) within an rectangle (instance of EventCallback). Suppose, in the example the activate message is sent to the event callback which holds a reference to the create port event. When the event callback receives the activate message, it calls voidCallback(), defined in the create port event. Each callback implements the callback interface thus implements this method. The interface code is given below. In voidCallback() the actual call to the operating system is made, which in this example is a method called create(), defined in the Ports module.

The operating system then handles the system call. After the system call has returned, a second message will be sent to the port event generator, which is the evaluate message. This results in the event generator putting a new event on the event queue. Then the whole process starts over again: the next event is taken from the event queue and an activate message is sent to the object referenced in this event.

How are events put on the event queue the first time? At the beginning of the simulation, after the event tree is build up, an evaluate message will be sent down the whole tree. This results in all sub trees putting the first events on the event queue. This is the only time a message is sent to the top object of the tree. During the rest of the simulation messages are only sent to objects in the sub trees.

2.2.2 Callbacks

A callback object implements the CallbackInterface, providing the event engine a standard interface to initiate the event.

```
public interface CallbackInterface
{
    public void voidCallback();
}
```

For every type of event simulated a new class has to defined which implements CallbackInterface as shown above. An instance of such a class can then be referenced by an event callback and this callback can be attached to the event tree.

The callback interface classes are defined in the IFL module. This way the interface class and the actual methods implementing the IFL part of the system calls can be defined in the same file. The callback classes are used by the IFL module to implement buildEventTree(), the methods called to build up the tree. In this method the callback objects are initialized, using the parameters from the parameter file. Then they are attached to the event tree.

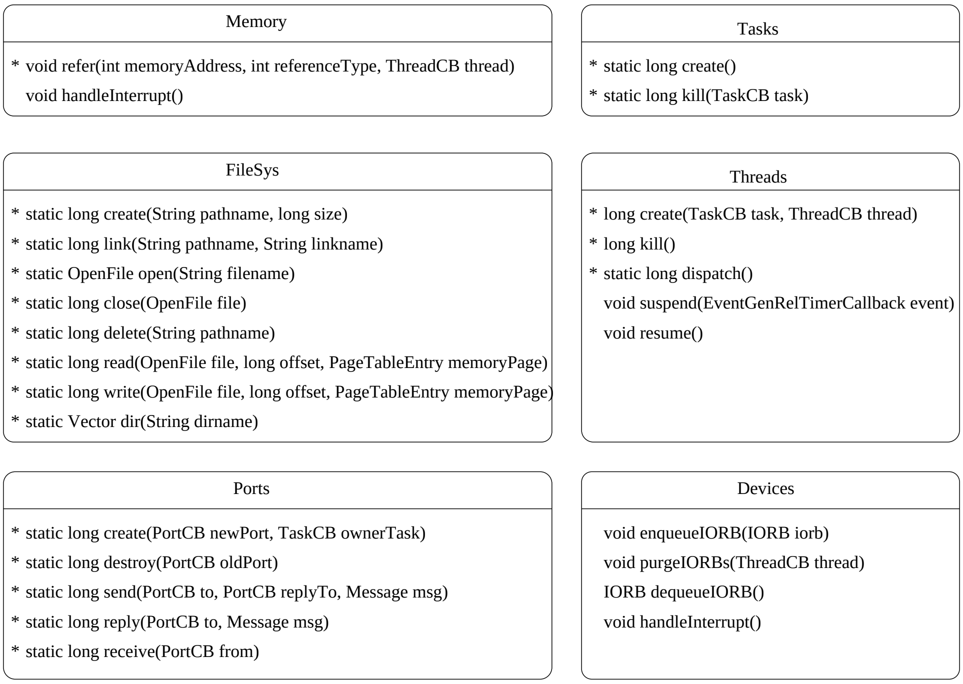

Figure 2.4: *Most important methods to be implemented by students. Methods with an asterisk (*) are called by a callback (system calls), the other methods only are called from the operating system itself.*

2.2.3 Distributors and generators

As stated before the difference between the several kind of objects in the event tree is in the way they handle activate and evaluate messages. Table 2.1 gives an overview of how messages are handled.

	activate	evaluate
EventGen	Pass message to all child objects.	Put a new event on the event queue and pass the message to all child objects.
EventDist	Choose one or more child objects and pass the message to this object.	Discard the message.
EventCallback	Call voidCallback() from the callback class referenced in this object.	Discard the message.

Table 2.1: *Message passing by object in the event tree.*

The difference between the several sub classes of EventDist is the way they one or more of the child objects are picked to pass the message. For example, in EventDistRndUniform a child is chosen to send the message to, every child having the same chance of being chosen. In EventDistRndUniform every child is associated with a certain probability of being chosen. This probability can be set by the programmer when the event tree is build up.
The differences between the subclasses of EventGen is in the way the time stamp is calculated when creating a new event. For example, an instance of EventGenInterval makes sure that the time between every event created by

this event generator is always the same. Other class might implement a more variable interval.
Now that all objects in the event engine are described, a summary is given of the actions taken by the event engine in one simulation cycle:

- The engine thread pulls an event from the event queue.
- It sends an activate message to the object referenced by this event (i.e. it calls the activate() method of the object).
- This object sends the activate message down the tree.
- The activate message is finally received by one or more event callbacks.
- These callbacks each hold a reference to a callback interface. The method voidCallback() defined in CallbackInterface is called in each of these callback interfaces.
- In voidCallback() a particular system call done, the IFL module takes over control.
- When the system call returns, the event engine takes over control again.
- The event engine sends an evaluate message to the same object as in the beginning of this cycle.
- The evaluate message is sent down the event tree which causes one or more event generators to put a new event on the event queue, having a time stamp which is higher than the previous one.
- A new event is taken from the event queue.
- And so on.....

2.2.4 Time

Each event is associated with a certain time stamp. The events are started in order of their time stamp. When an event is taken from the event queue a global time field, defined in GlobalVariables, is updated with the time stamp. This is how time increases in OSP.
When a new event is put on the event queue, a new time stamp is calculated by the object creating the new event. The new time stamp is calculated from the previous time stamp as well as from variables defined in the object. These variables are set at the beginning of the simulation, when creating the objects in the event tree. They are determined by the parameters defined in the parameter file. As an example, suppose relatively more tasks should be generated. In that case, the difference between the time stamps in the old and the new event for creating a new task will be kept small. This way more tasks will be created during the simulation.
The new time stamp is not only calculated from the old time stamp and the variables defined in the object creating the new event. In some cases also a randomly chosen number is used in order to keep the time between events variable, resulting in a more realistic system.
When the global time variable reaches a certain value, which is defined in the parameter file, the simulation ends. This way the student can specify when the simulation should stop.

2.2.5 Creating events from the event queue

The event engine is a nice way of simulating events. By building up the event tree the OSP programmer controls the way simulation is done. Also, it is easy

to implement a new kind of event generator or distributor. The only thing to do is create a subclass of EventGen or EventDist which overloads the activate() and evaluate() methods. It is a bit harder though to fully understand the event engine. The functionality of the event engine seems a bit overdone, since in OSP only a relatively simple event tree is build up. Also there is a strange way of adding interrupt events to event queue, which will be explained next.
How are new events created by the operating system? For example, if an I/O request is done, then an I/O interrupt should be simulated at a certain time in future. This means that an event has to be put on the event queue by the IFL module.
A solution to this problem, which was implemented in OSP, is creating an instance of class EventGenRellTimerCallback or one of its sub classes. EventGenRelTimerCallback is a subclass of EventGenRelTimer (see figure 2.3). If an object of this class receives an evaluate message the first time, it puts a new event on the event queue. But the second time it will just discard the message.

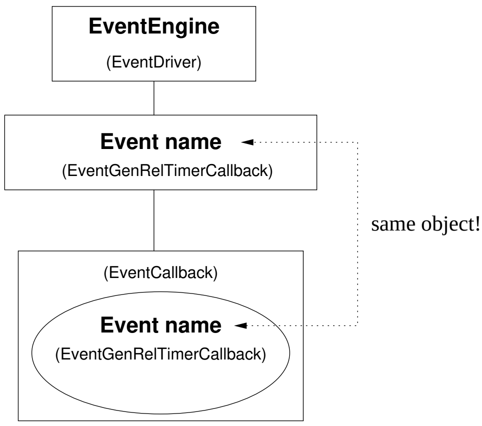

Figure 2.5: *Sub tree for simulating a single event*

If the OSP programmer wants to have just one event simulated, for example a disk interrupt, he or she creates a new class which extends EventGenRelTimerCallback. By overriding voidCallback() the programmer can determine what should be done when the event is pulled from the event queue.
The strange thing about this construction is that the EventGenRelTimerCallback adds itself to its own child list and also attaches itself to the event tree. This is depicted in figure 2.5. It then calls evaluate() which puts a new event on the event queue. When after a while the event comes of the event queue, the event engine calls the activate method of the newly attached object. This calls the activate method of the event callback, which in turn calls the voidCallback method of EventGenRelTimerCallback.
This is a strange way of adding a single event to the event tree. It is not necessary to attach a new object to the event tree and use this strange class EventGenRelTimerCallback. It would be better to make a new class which implements the callback interface and does not extend the EventGen class, just like with the other events. By providing a method in the event engine, which allows the programmer to put a single event on the event queue the problem can be solved as well, in a way which is much more comprehensible.

2.2.6 Java threads

It is important to realize that operating system code is executed by OSP user threads. making it possible for an OSP thread to put itself to sleep while executing operating system code. In systems where operating system code is executed outside all threads, this would not be possible, because the state cannot of the process cannot be saved then, thus it could not be resumed.
In OSP this is possible though. A thread can put itself to sleep and, for example, wait on I/O to finish. But how to simulate this in a realistic way? When a thread puts itself to sleep other threads should run. After a while the thread will run again, starting execution at the same place as were it was put to sleep.
The solution chosen for OSP is to create a new Java thread each time when certain interrupts or system calls will be simulated. This new Java thread will be associated with the OSP thread and starts the event. When at a certain point the OSP thread puts itself to sleep, as implemented by the student, the Java thread will suspend itself. At this point the normal engine threads starts pulling of new events from the event queue.

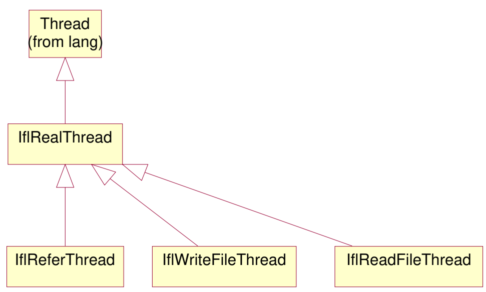

Figure 2.6: *Class hierarchy of OSP Java threads. Attributes and methods are left out.*

At a certain moment the event for simulating the I/O interrupt is taken from the event queue. The callback function calls the disk interrupt handler which then resumes the suspended OSP thread. The IFL module makes sure the associated Java thread will also be resumed.
The student does not have to know anything about these Java threads. The IFL module takes care of suspending and resuming the Java threads. The student only suspends the OSP threads. But when an OSP thread gets suspended, the IFL module will suspend the Java thread.
There are four cases in which a thread can be suspended. The first one is when the MMU interrupts the processor and the page fault handler must be called. This can result in swapping in a page, thus in I/O. When a thread has to wait on I/O it should put itself to sleep, allowing other threads to run in the mean time.
The next two cases are when reading or writing a file. This also results in I/O. The fourth case is when sending or receiving a message to or from a port. In

these cases it is possible that a thread has to wait until a message is received or sent and also has to put itself to sleep.
For each of these cases a sub class of IflRealThread is defined. IflRealThread is a subclass of Thread, defined in the standard Java library. In each of these four cases the engine thread will not issue the system call or start the interrupt itself, but create a new Thread, an instance of one of the four sub classes. The new thread will then start the interrupt or issue the system call.
By using a thread lock it is assured that the Java threads will not run concurrently. The thread lock is part of the event engine. When the engine thread creates a new Java thread, it passes the lock to this new thread. The engine thread will then have to wait until the new thread releases the lock, for example when going to sleep. In that case the engine thread will become runnable again and starts pulling of events from the event queue again. Until the event comes up which resumes the other thread again. In that case the event engine passes the lock to the other thread again. This way control is kept over the Java threads, not allowing the Java thread scheduler to run the threads concurrently.
The use of Java threads makes the code in the IFL module more complicated but also much more realistic. The student will notice that OSP threads actually stop executing when going to sleep and starting at the same place as were they were left when resumed again.

2.3 Hardware and Interrupts

The Hardware module simulates the hardware on which the operating system is running. The simulated consists of a processor, a memory management unit (MMU) and several disks.
The operating system accesses the hardware for two things. The first is to initiate disk I/O, the second to specify the task being executed by the processor. The hardware also generates interrupts (initiated from the event engine). An interrupts is passed to the interrupts module, which is actually part of the operating system. The interrupts module is responsible for calling the correct interrupt handler.
Three classes are defined in the hardware module: CPU, Disk and Machine.

Machine As can be seen in figure 2.7, the machine keeps a references to all parts of the hardware. Strangely enough, it also has a reference to a task. This is the task currently running on the CPU. This value should be stored in the operating system though, not in the hardware.
Secondly, a machine also holds a number of frameTableEntries. These also should not be part of the machine. A frame table is stored in the operating system. All a machine has, is a number of frames, containing data. But because one of the simplifications in OSP is that no data is stored these references can be left out.

CPU The CPU has two attributes which are not used in the current implementation of OSP, so they will not be discussed. Method interrupt() is called from a disk or the MMU to interrupt the CPU.

Disk A disks consists of a number of platters. Each platter contains a number of blocks. As can be seen in figure 2.7 Disk extends Devices. A device is stored in the operating system and should hold a reference to a disk in hardware in

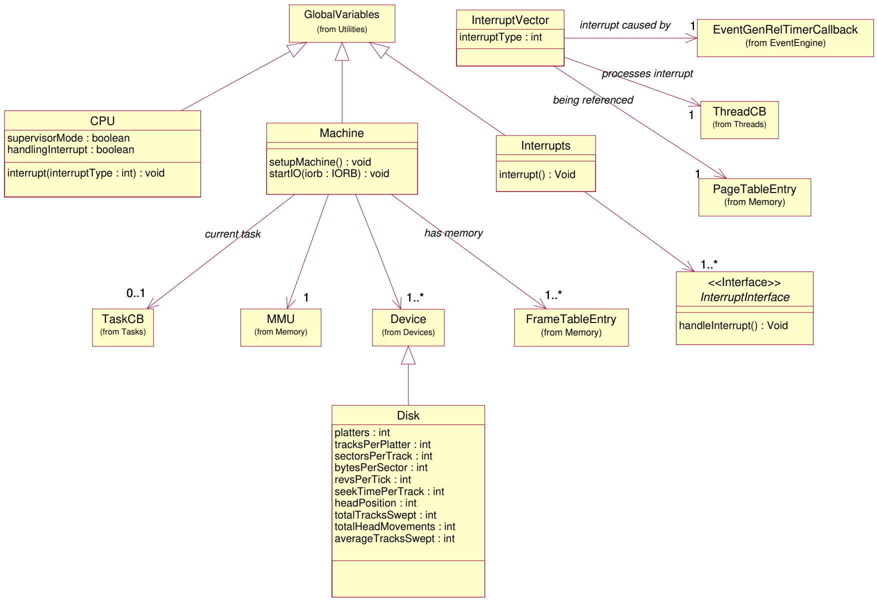

Figure 2.7: *Classes in Hardware and Interrupts*

stead of subclassing Device. In the current implementation, hardware holds references to a number of devices. Better would be to have disks in hardware and add an reference to a disk in each device.
Though the MMU is actually part of the hardware, it is implemented in the memory module. It is part of the assignment from the memory module. The MMU will be discussed in section 2.4.4.
The Interrupts module also defines three classes: InterruptInterface, InterruptVector and Interrupts.

InterruptInterface This class describes the interface implemented by the interrupt handlers. It is implemented by two classes: IFLDiskInterruptHandler and IFLPageFaultHandler.

InterruptVector This class contains some variables which are set before calling an interrupt handler, so the interrupt handler can see why the interrupt was generated.

Interrupts Interrupts stores the two interrupt handlers. These can be started by calling interrupt().

2.4 Operating system classes

Next a description will be given of the classes which form the actual operating system. Classes in both the students module and the IFL module will be described for each of the modules Tasks, Threads, Ports, Devices, Memory and

FileSys. Some details are left out in order to make it easier to get the whole picture. Not all classes, methods and attributes are mentioned. Most access (get() and set()) methods are left out. The design should give a complete picture of how the OSP operating system is currently implemented. A class diagram and object collaboration diagram are given in appendices B.2 and B.3.
The reader should keep in mind that OSP2 is still under development. The diagrams show how things are currently implemented, but this is not the final version. Some details still need some improvements. Especially the way attributes are divided among the IFL and students module might lead to some questions. The reason is that several programmers have worked on the program, leading to differences in the way of doing error checking.
The main focus will be on the functionality of the operating system though. Error checking is mainly a question of implementation: where should attributes be stored to do proper error checking. In this chapter, mainly the operating system as a whole will be discussed, giving suggestions for improving the design of the operating system. These suggestions will also be used in chapter 4. Therefore, most of the time a class defined in the students module will be discussed together with the associated class in the IFL module.

2.4.1 Tasks

A task can be compared with a Unix process, only it is much simpler. A task is a container for threads and resources. A task can be created and killed using system calls.

TaskCB and IflTaskCB A TaskCB (Task Control Block) has a number of threads of which one might be running on the CPU. These threads all have access the resources from the task. The resources are a page table, a swap file, a list of open files and a list of ports.
Figure 2.8 shows the class diagram. The page table and swap file have to be accessed via the IFL module, threads and ports have to be stored in the students module. Notice that an attribute with a name beginning with Ifl is not visible to the student. Such attributes are only used to do error checking. The operating should also work without these attributes, only no error checking would be done then.
The create() system call initializes all resources, the kill() system call removes them again. Other methods are used to add or remove a port or thread to the list of resources of the task. These methods are called from the Threads and Ports module.

2.4.2 Threads

A task can have several threads. Threads execute on the processor. In OSP this means that the current task in Machine is pointing to the threads owner task, and the current thread of the current task is pointed to the thread. A thread can be running, waiting, blocked, or killed. Methods are provided to change the current state of the thread into another. Threads can be created and killed by the simulated software. When a thread has to wait on I/O, it will suspend itself. The disk interrupt handler is responsible for resuming the thread again after the I/O has taken place.

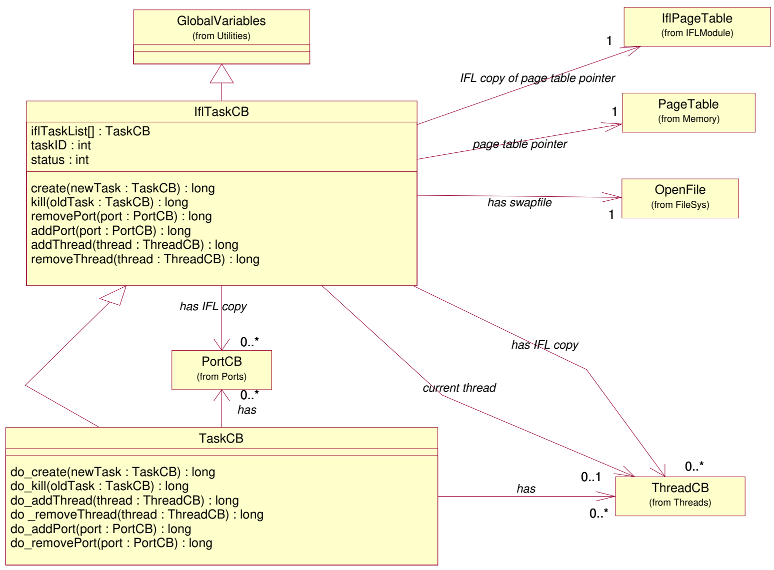

Figure 2.8: *Classes in Tasks module*

When a thread is created its state will be set to ready and added to the ready queue, which is a static list of threads. The thread can now be dispatched. A running thread can be suspended on a certain event. It will then be added to the thread list in the event. When the event happens, one or more threads from this list will be put in the ready queue again. Like in TaskCB, only one class in the students part of this module is defined, which is ThreadCB (Thread Control Block).

ThreadCB and IflThreadCB Figure 2.9 shows the threads class diagram. Most attributes are stored in the IFL module. A thread has one owner task, which stays the same while the thread is alive. The message field is used by the Ports module.

A thread can be blocked on three kind of event. When it is waiting for a disk to read or write it is blocked on an IORB (I/O request block). This is a sub class of EventGenRelTimerCallback defined in the event engine. A thread can also be blocked on a port (this will be explained in section 2.4.3). PortCB is also a sub class of EvenGenRelTimerCallback. When a thread is waiting for a page to be swapped (which is handled by another thread), it is blocked on a PageValidationEvent. Both these events can be put on the event queue from the event engine. No distinction is made between events within the operating system and events from the event engine. Better would be to have two separate kind of events, one used only in the event engine, which can put in the event queue. The second can be used within the operating system and only holds a list of threads which are suspended on the event.

An IflRealThread object can be attached to the thread, to simulate the suspending of the thread. This is explained in section 2.2.6. The IflMMUSnapshot is used to do error checking (see section 1.3.3). Another attribute which cannot be accessed from the students module is the ResumeThreadEvent object. It is

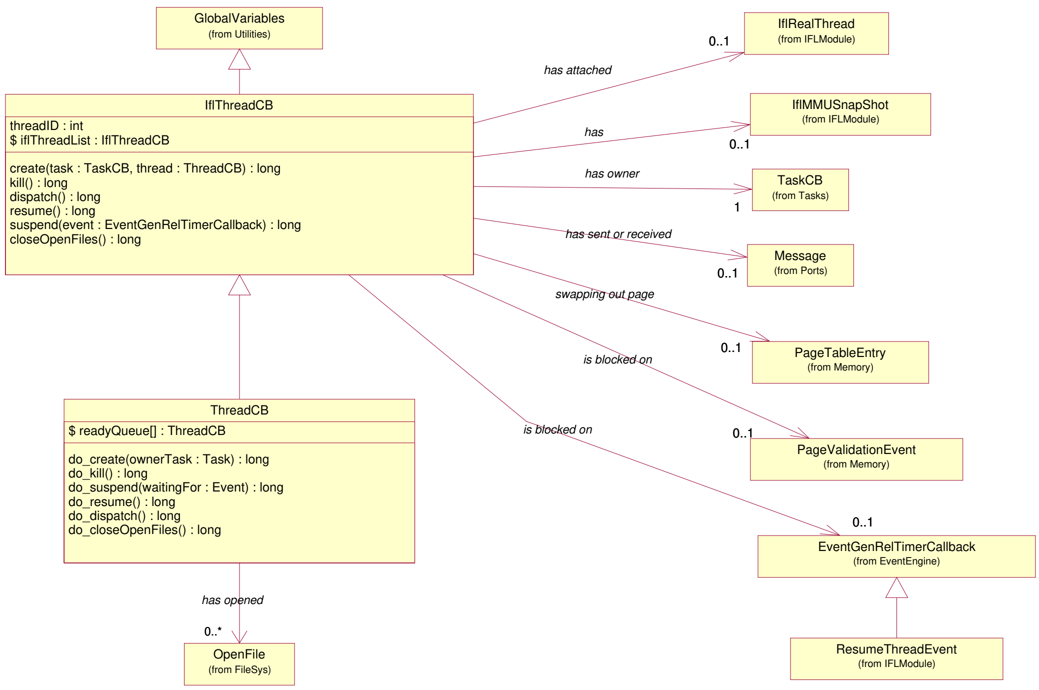

Figure 2.9: *Classes in Threads module*

used to resume the Java thread after the OSP thread is resumed.
A thread can be created or killed using the system calls create() or kill(). Suspend() and resume() are used for suspending and resuming the thread. Dispatch() is called in case of a timer interrupt. It is directly called from the event engine, not from the hardware. More realistic would be to have the interrupt generated via the hardware.

2.4.3 Ports

Ports allow threads to communicate with each other. Ports are used to send and receive messages. A task can hold several ports and receive messages via that port. All threads from the task have access the tasks ports. Messages can be received via that port. Other ports, owned by other tasks, can only be used for sending messages. When a thread wants to send a message, it will store the message directly in the port of the receiving task, thus not via one of its own ports.
A thread can be blocked when sending or receiving a message. When a thread does a receive() system call on a port which contains no messages, it will be suspended on the port. When it wants to send a message to a port with a full buffer, the thread also will be suspended. A thread will be resumed again when another thread wants to send or receive a message to or from the same port.
Two classes are defined in the students module, PortCB (Port Control Block) and Message.

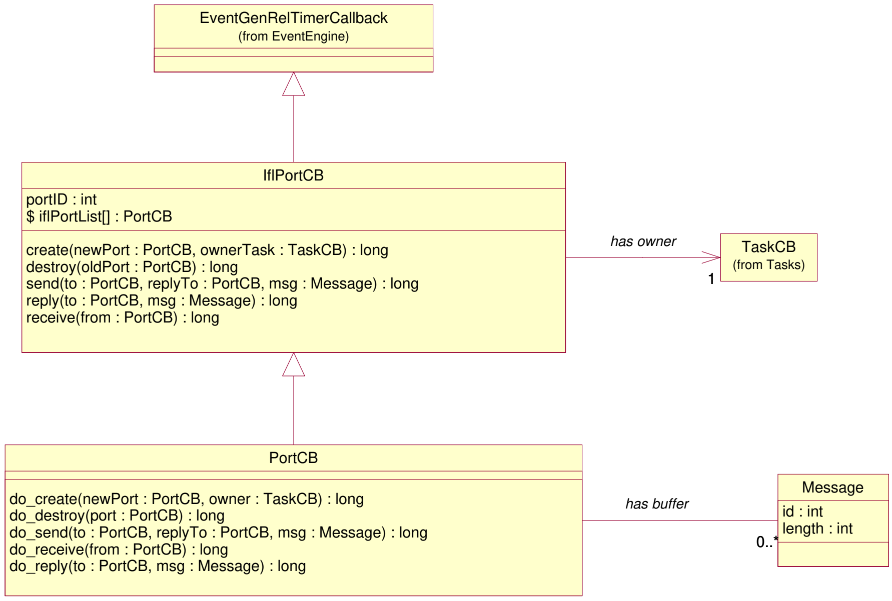

Figure 2.10: *Classes the in ports module*

PortCB and IflPortCB A port has a buffer in which message are stored. This buffer can only a store a maximum number of bytes. Messages can be of variable length, thus the number of messages which can be stored in a port is not fixed.

Message No actual data is stored in a message. Only the length of the message is important.

2.4.4 Memory

In OSP, virtual memory is provided. In the memory module pages and frames are stored. Pages are stored in page tables. Each task owns one page table. A page can be in memory, thus stored in a frame, or not. In that case the page is only stored in the swap device. The memory module is responsible for keeping track of where pages are stored. When a pages needs to be accessed by the simulated software, the MMU will be referred. The MMU will check to see if the page is in memory (stored in a frame). If not, an interrupt will be generated. The page needs to be swapped in from the swap device. The page fault handler is responsible for doing the swapping.
Next, all classes in Memory will be discussed. More about swapping in OSP will be explained in section 4.4.6. Most attributes are stored in the IFL module, taking away responsibilities from the student.

MMU and IflMMU The MMU class simulates a part of the hardware, though it is stored in Memory, because it can be used in an assignment. The

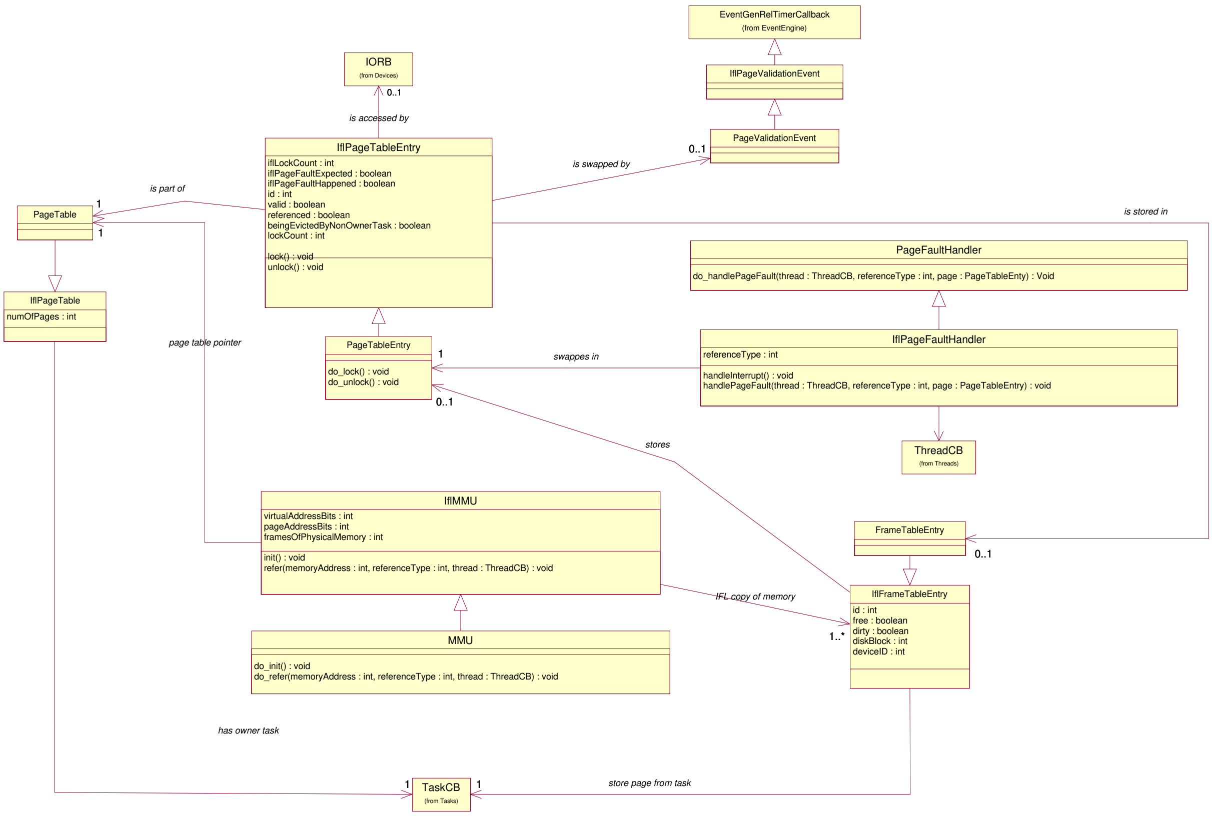

Figure 2.11: *Classes in memory module*

student has to implement the method refer() which checks to see if a certain memory address is valid or not. If not, an interrupt has to be generated.

FrameTableEntry and IflFrameTableEntry The FrameTableEntry class contains information about a specific page frame in memory. A frame table entry consists of number of flags which reflect the state of the frame and a reference to the page the frames currently stored.

PageTableEntry and IflPageTableEntry A page table entry can be stored in a frame table entry. If it is the page is valid. It is part of a page table and can be swapped in or out of memory. When the page is being swapped out, allowing another page to be swapped in, the dirty flag has to be checked. If the page is dirty, thus it has changed, the page should be written to the swap device. Otherwise it can be overwritten.
Before reading or writing from or to a page, the page needs to locked. This way no other devices can access the page. A page can be locked using the lock() method. This method increases the lock count and makes sure the page is valid. If the page is not valid, the page fault handler will be called. Unlock() will release the page again. This method is called from the device module.

PageTable and IflPageTable Each task holds a page table. A page table consists of a number of page table entries. This number is fixed, it is the same as the number of frames in memory.

Pagefaulthandler and IflPagefaulthandler This class implements the InterruptsInterface and its method handleInterrupt() is called when a memory reference is made to page which is not in physical memory. In that case the holding the reference virtual memory address needs to be swapped in. The page can be read from the swap file which is stored in each task. Using the read() method from the file system, the page can be written to memory.

PageValidationEvent and PageValidationEvent A PageValidationEvent is used to suspend a thread. When a page swap is started, the pageValidationEvent is added to the page. When starting the I/O for swapping in the page, the thread will be suspended, thus it will be added to the list of threads in this pageValidationEvent. Once the page is in memory, the threads from the list of threads will be resumed.

IflMMUSnapshot This class is used to do error checking. An instance of this class is attached to a thread when a page fault is generated. By storing data about the page fault in this object, the IFL module can check whether or not the student handled the page fault correctly.

2.4.5 Devices

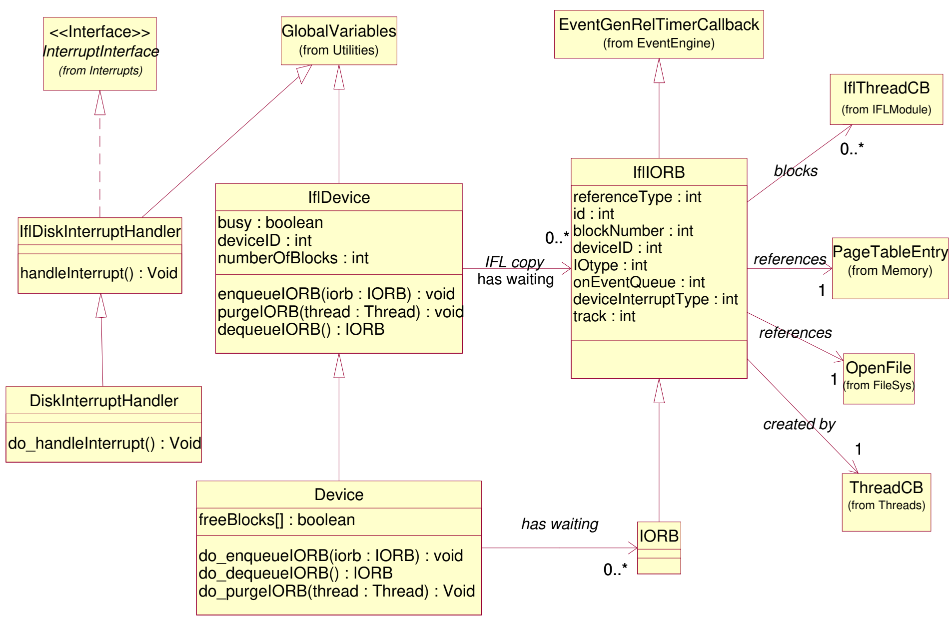

Figure 2.12: *Classes in device module*

A device acts as the interface between disks from hardware and the rest of the operating system. An IORB (I/O request block) can be sent to a device

by the from the file system. The device will then process this IORB. If the disk associated with the device, the IORB will be stored in the IORB queue. Otherwise it will be processed directly. I/O will be started by calling startIO() defined in Machine. The IFL module will put an event on the event queue in the event engine to make sure a disk interrupt will occur later. In the mean while, the thread which started the I/O will be suspended.
When the disk interrupt is generated, the disk interrupt handler comes in action. The disk interrupt handler updates the structures in the disk and resumes the threads waiting for the I/O. Then the next IORB can be processed.

Device and IflDevice The size of the block is equal to the size of the memory page. A device processes the IORBs which are queued in the device by the file system.

IORB and IflIORB An IORB object only stores information about a particular IO request. All attributes are stored in the IFL module, leaving the students module empty. No methods are defined. IORB is a sub class of EventGenRelTimerCallback making it possible add an IORB to the event queue. A thread can be suspend on an IORB. (see 2.4.2).

DiskInterruptHandler When a disk I/O interrupt occurs, handle_interrupt() is called. This method is responsible for resuming the threads waiting for the I/O to be finished.

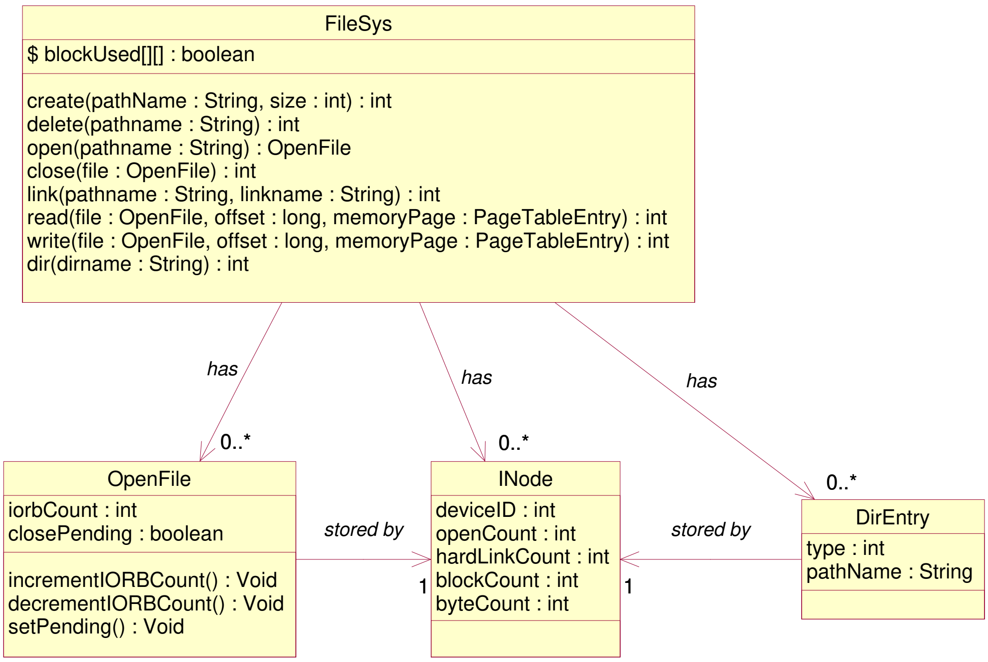

Figure 2.13: *Class diagram of the FileSys modules.*

2.4.6 FileSys

The file system will not be discussed in much detail in this document. This should not be a problem for understanding the rest of the system though, because collaboration between the file system and the rest of the system is very limited. The section is added to complete the picture of the current design of OSP.
The main functionality the file system provides to the rest of the operating system is used for swapping. Pages need to be swapped between the swap device and main memory. So the most important methods to rest of the operating system are read() and write(). Most of the other system calls, like link() and create() do not need to communicate with the rest of the operating system at all. These methods only update the internal data structures of the file system. Create() and delete() are used to create of remove a file or directory. Open() and close() for opening and closing files. Dir() returns a list of all files and directories in a certain directory. Link() can be used to create a hard link.
More about the file system can be found on the web site of OSP:
www.cs.sunysb.edu/˜osp++.

2.5 Conclusion

In this section the design of the current OSP was given. It turned out that the hardware, the operating system and the event engine are not totally separated modules. Events used in the event engine are also used in the operating system. Disk is a subclass of Device, so actually a part of the operating system is stored in hardware.
Though a choice was made to use Java threads so the suspending of threads could be simulated better, in some other parts of the system the reality of the simulation was not taken into account. Changes in the definition of the classes leading to small changes in the implementation can result in a more realistic system.
Especially in the Memory module, a lot of responsibilities are stored in the IFL module. Most classes in the students module hardly need any implementation. This will not result in a system that seems realistic to the student. The students part of the operating system is kept very simple.
In the next chapter a closer look at existing SMP systems will be taken. Then, in section 4 a design will be given for SMP OSP. In the new design the main focus will be on the operating system, trying to make it a more realistic one.

3 SMP systems

In order to make a realistic SMP OSP design, some knowledge about SMP systems is necessary. In this chapter a closer look will be taken at SMP systems. First a classification of parallel computers will be given. Then hardware and software of SMP systems will be discussed.

3.1 Parallel computers

A lot of investments have been done in developing parallel computers trying to increase the computational power of a single processing unit. All kind of parallel computers exist. To make clear what categories of parallel computers can be defined, a taxonomy given by Flynn [11] will be used. This taxonomy can be found in almost every book about parallel computers and the terms Flynn defines are still used a lot. It is based on the number of instructions and data streams that are executed simultaneously. Stream in this context means a sequence of items (instructions or data) as executed or operated on by a processor.

- *SISD* (single instruction single data) stream: A single instruction stream executes on a single stream of data. A uniprocessor system falls into this category.
- *SIMD* (single instruction multiple data) stream: Multiple streams of data are operand of a single stream of instructions. Examples are vector and array processors.
- *MISD* (multiple instruction single data) stream. Multiple instruction streams process a single stream of data. No implementation is found to serve as an example.
- *MIMD* (multiple instruction multiple data) stream. Multiple streams of instructions are executed on a multiple stream of data. Most multiprocessor systems fall into this category.

MIMD is the category which will be discussed here. Within the MIMD category other categories can be defined. One way of doing this is by looking at the granularity (the frequency of communication) processors expose. A second way is by looking at how processors are connected to the memory modules.
In tightly coupled systems the processors are physically close together (for example on one motherboard) and the communications frequency is high. The processors share a certain memory although each processor can have its own memory as well. The processors can communicate by sending each other direct signals, but most communication is done via the shared memory.
In loosely coupled systems each processor has its own memory and the frequency of communication is lower. No processor has immediate access to the memory

of other processors. Communication is only possible by sending messages. If a processor wants data from remote memory it must ask the processor holding the memory to send the data (server/client model). An example of loosely coupled systems is a cluster. This is a set of stand alone computers which communicate via fixed paths or via some network.

3.1.1 Memory

Tightly coupled systems can be further subdivided by the way access to memory is implemented. In a UMA (Uniform Memory Access) system, each processor will be able to retrieve any word in memory with the same speed. While in a NUMA (Non Uniform Memory Access) system the access speed to memory modules do not have to be the same for each module. In this case it is important that the operating system looks at where data can be stored best. If a processor performs a lot of operations on a particular set of data, it is best to place this data in that part of memory which provides the fastest access to this processor.

Memory is often organized in such a way that simultaneous access is possible to separate blocks in memory. There are several ways of connecting memory modules to the processors in physical hardware [9]. One way is using a bus. All memory modules and processors are connected to the bus. All data a processor needs will be transported over the bus. The bus can become a major speed limit, because all processors use the same bus. For a system with two processors this will not be a big problem, but for a system with 64 processors this will not do.

A solution would be to use a crossbar switch network. This is a non-blocking network which means that if a processor wants access to a memory module which is not yet occupied, access will be granted. The advantage is the increased speed in memory access. The disadvantage is that a crossbar switch network is a lot more expensive. The number of switches needed is the same as the number of processors times the number of memory modules.

Another solution is a (multistage) switching network. This is a blocking network, but it is cheaper because less switches are needed. The number of stages and switches per stage determines the access speed and costs of such a network.

In both networks it is important to spread memory references a much as possible in order to prevent as much as possible processors wanting access to the same memory module at the same time.

3.1.1.1 Cache

One method of speeding things up is by providing each processor with a certain amount of cache memory. The cache memory keeps a small copy of the shared main memory. Not only is access to cache memory faster, it also prevents the processor from occupying the bus or network.

A problem arising is the cache coherence or cache consistency problem. This is the problem of how to keep all data consistent. When a processor updates a certain part of its cache memory, how do the other processors know that this update has taken place? Protocols are designed to prevent this problem from happening (see [10]). These protocols are implemented in hardware and are not an issue in operating system design.

3.1.2 Software

Operating systems running on multiprocessor need some extra functionality opposed to operating systems on a uniprocessor system. The operating system must provide the following things[9]:

- hide the multiprocessor implementation from the user. The existence of multiple processors is invisible to the user, the operating system takes care of dividing the work among the processors.
- take as much advantage from the processors as possible. The operating system must exploit the parallelism as good as possible.
- provide higher reliability. When one processor fails, the other processors can take over. Some operating systems provide graceful degradation, making sure that the system will keep running correctly when a processor fails.

3.2 SMP

One way of classifying tightly coupled systems is the way kernel responsibilities are divided among the processors [4]. In a master/slave system, one processor is responsible for doing all kernel activities. This is the master processor. All the other slave processors ask the master processor which process they should execute. Only the master processor has access to all the kernel data.

In a symmetric multiprocessor system (SMP) each processor executes kernel code. All the processors have the same responsibilities, hence the term symmetric. The kernel can be executed by any processor and each processor does self scheduling. Parts of the kernel can be executed in parallel, by separating the kernel into several modules.

These are two extremes within tightly coupled systems. Unlimited possibilities exist in between. The hardware of the system determines how much symmetry is possible. It is possible that not all processors have access to certain IO devices. In that case the processors which do have access are responsible for the device handling. So symmetric multiprocessing refers to a computer hardware architecture as well as to the operating systems behavior. In true SMP systems, the processors share all the memory as well as the I/O facilities [8].

Symmetry can exist in two parts of the hardware, memory and I/O. Memory is symmetric when all processors share the same memory space and access that space by the same addresses. Memory symmetry allows all processors to execute a single copy of the operating system.

Symmetry in I/O means that all processors have access to the all the I/O devices. All processors can receive interrupts from all devices. Some multiprocessor systems are symmetric concerning memory access, but not concerning the I/O devices. Then, one or more processors are dedicated to handling the interrupts.

Both extremes have their own characteristics [4]. A master/slave operating system is simple to implement. It does not differ much from a uniprocessor operating system. One problem though is that the master slave can become a bottle neck. All the other processors will do requests to the master, which the master might not be able to process fast enough. This will probably not be the case for a system with only 2 processors, but when the number of processors increases the load on the master processor will increase. So, master/slave systems are less scalable than symmetric systems.

Another problem with master/slave systems appears when the master is failing. In this case the whole system will fail. This problem brings down the reliability of the system. A master/slave system can be useful when one processor is much faster then the others.
An SMP system requires a more complex operating system. Because all processors have access to the kernel synchronization methods are needed to keep the kernel data valid. But an SMP system does not show the characteristics a master/slave system has, as mentioned above. It scales better and can give a higher reliability.
Some changes have to made to turn a uniprocessor operating system into a multiprocessor operating system. The main changes are the following [4]:

- *Scheduling:* processes and threads must be scheduled over several processors.
- *Data consistency*: several processors might run kernel operations at the same time, i.e. the kernel is reentrant. The operating system must make sure that all the kernel data stays valid by using synchronization methods.
- *Memory management*: The operating system must make sure all the page tables stay correct. It is possible that two processors run two threads from the same task, thus sharing the same page table. When the MMU uses a table lookaside buffer (TLB) the operating system must make sure that all TLBs stay correct when pages are being brought in or shifted out of physical memory.
  If memory is divided into several modules, the operating system must try to make sure as much as possible that processors do not try to access the same module at the same time. Thus it must try to spread memory access. In the case of a NUMA system, the operating system must carefully choose where to store data.
- Graceful degradation: The operating system makes sure that the system will keep running correctly in the case of failure of one of the processors. This functionality will not be discussed any further.

3.2.1 Interrupts

In a true SMP system the I/O devices are all accessible by all the processors. Most SMP systems have the ability to direct interrupts to certain processors. This way it is possible to perform load balancing. For example, by making sure that an interrupt will be sent to the processor which is idle, other processors can continue doing their job.
In some systems each processor has its own clock and thus every processor will receive a clock interrupt separately. In other systems there is only one clock which sends an interrupt to one or more processors. The processor handling this interrupt can use interprocessor interrupts (IPIs) to send a signal to other processors. IPIs make it possible for the processors to communicate with each other without using the shared memory. IPIs are called soft interrupts because they are generated by software.
Interrupts can be categorized in directed, undirected and broadcast interrupts. A directed interrupt is an interrupts that is always posted to the same target processor. The processor that receives an undirected interrupt can change during the execution of the system. The choice where to send a certain interrupt to can be made by using a distribution scheme or the operating system can

program the interrupt hardware and tell it to send certain interrupts to a specific processor during the running of the system. A broadcast interrupt is an interrupt that is posted to all processors in the system.

3.2.2 Memory

There are systems in which processors have faster access to certain parts of memory (NUMA). This is not a true SMP system. Some SMP operating systems do not take this hardware characteristic into account and treat it as an UMA system. In this section UMA memory will be discussed.
An important change in the memory management of an SMP operating system is the management of the page tables. It is possible that two processors execute two threads from the same task. When a certain thread asks for a page to be swapped in or out, the operating system has to take into account that other processors might access the same page. It is possible that using the uniprocessor memory management system, a page which is used by a thread on another processor is swapped out. Thus the replacement algorithm, which decides which page to swap out when another page is to be swapped in, might need some changes.
Another thing to think about is the use of a table lookaside buffer (TLB). In order to keep all TLBs up to date the hardware must provide some functions to flush the TLB of a certain MMU. This can be done by using interprocessor interrupts. When a processor in kernel mode swaps a page, it must tell all MMUs which hold references to this page, to update the TLB.

3.2.3 Synchronization

A big difference between an uniprocessor system and a multiprocessor system is the true parallelism. In a uniprocessor system, processes seem to run in parallel, but this parallelism is simulated by interleaving several processes. In multiprocessor systems, processes run in parallel, thus processors can try to enter the kernel at the same time. This asks for some changes in the operating system.
In order to keep processors synchronized, the hardware must provide instructions which allow the programmer to implement synchronization methods. Some processors have test and set instructions or swap instructions. These instruction allow the programmer to read and then write a certain word or byte in memory without being interrupted. Reading and then writing a word is executed as a single atomic instruction. Using these instructions processors can communicate with each other and the programmer can guarantee mutual exclusion.
There are three ways of implementing a process waiting for a semaphore or other kind of synchronization method [8].

- *Busy Waiting.* With busy waiting, the processor continuously tries to acquire the lock by checking the shared variable which is used by the lock over and over again. This is also called spin locking. The processor keeps trying to access the lock and thus keeps using the network which connects the processor to the memory. This way other processes running on other processors will run slower, because of the reduced access speed to main memory.

- *Sleep-lock.* In sleep-lock a processor is suspended, when it does not get access to the lock. It simply stops executing. The processor can only be activated again by an interprocessor interrupt, telling the processor the lock has just been freed. Then the processor can try to get hold of the lock again. This method reduces network traffic, but still wastes processor cycles.
- *Queuing.* In this method, a process trying to acquire a lock which is already used is put on a global queue. When the lock is released the next process from the queue is given the lock. This introduces some extra overhead, because of the queuing and dequeuing of the process. The queue has to be protected against concurrent access.

The programmer has to be careful in synchronizing the threads. There are some known problems among which the lost wakeup problem and the thundering herd problem [12]. The lost wakeup problem is about threads waiting for a resource which is not locked anymore. Suppose a thread wants to acquire a resource, but the resource is locked. The thread will go to sleep but the testing of the lock and going to sleep is not an atomic operation. The lock might be released between the testing and going to sleep. When the lock is released, threads waiting for the lock are resumed. But the thread did not put itself on the list of waiting threads yet. Thus it will put itself on the list after the lock has been released. This results in a thread waiting for a resource which is not locked.
The thundering herd problem is about waking up all threads waiting for a resource. When a resource is locked and all threads waiting for the resource are resumed, they will simultaneously try to acquire the resource. In a uniprocessor this is hardly any problem, because only one thread at a time can acquire the resource. But in a multiprocessor system all threads can try to acquire the resource in parallel, and only one will get it. Thus a lot of time is wasted in resuming and suspending the threads.
To overcome these problems several kind of synchronization methods can be used, like semaphores and events. More about these methods can be found in [12].

3.2.4 Scheduling

The choice of which scheduling algorithm to implement should mainly depend on the following things [8]:

- degree of granularity, i.e. the frequency of communication the processes show
- number of processors
- use of cache memory

In coarse and very coarse grained systems the scheduling algorithm does not need much change to run in a multiprocessor system. In medium-grained parallel systems, where parallel processing or multitasking within a single application exists, scheduling is important because threads in such systems interact frequently. When two threads communicate a lot, it is best to have them dispatched at the same time, so that a thread does not have to wait long for a reply.
This is why it is important to make a distinction between process scheduling and thread scheduling. Because processes do not communicate as much as threads within a certain task do, thread scheduling can have much more impact

on performance. Research has shown that process scheduling has less impact on performance in a multiple processors than in a single processor system. A simple FCFS (First Come First Serve) discipline will often perform better than other complicated algorithms. In this section mainly thread scheduling will be discussed.
A second system characteristic from the list is the number of processors used in the system. Whereas with two processors it is important to keep the processors busy as much as possible, when much more processors are used, it can sometimes be better to leave a processor idle for a certain amount of time in stead of running the next thread available. Some threads may run poor if not all of its threads are running simultaneously. So, the kind of applications running on the system also influent the performance.
The third item on the list is the use of cache memory. When every processor has its own cache memory, it can be important to keep a process running on the same processor as much as possible. If the process is moved to a different processor every time, a lot of time would be spend to load data from main memory into the cache memory. If the process is run on the same processor every time, chances are higher that the cache memory still holds the parts of memory used by this particular process. This is called the cache corruption problem and it can significantly degrade performance.
Another thing to keep in mind is that context switching can result in a lot of overhead. A large number of instructions need to be executed, in order to save and restore registers.
A distinction between static and dynamic scheduling can be made [4]. In static scheduling, a process or thread has a dedicated processor. Apart from avoiding the cache corruption problem, there will be less overhead in the scheduling function. A disadvantage is that one processor can be idle while another processor's queue is not empty. This is not the case with dynamic scheduling, where a process can be executed on several processors during its lifetime.
Among all multiprocessor thread scheduling a number of approaches can be recognized, which will shortly be described (see [4] for more information about multiprocessor scheduling).

- *Load sharing.* In this dynamic scheduling algorithm, a global queue of ready threads is maintained, and each processor when idle, selects a thread from this queue to run.
- *Gang scheduling.* A set of related threads is scheduled to run on a set of processors at the same time. This way a thread which wants to communicate with another thread of the same task does not have to wait for the other thread to become available, thus reducing synchronize blocking. Also scheduling overhead will be less, because a single decisions affects multiple processors.
- *Dedicated processor assignment.* Each program is allocated a number of processors equal to the number of threads in the program, for the duration of the program execution.

3.3 Hardware implementations

To get an idea of how real multiprocessor systems are designed, next two examples of SMP hardware implementations will discussed. Only a few things from both systems will be highlighted. The systems are the SPARC specification (see [7]) and the x86 series from Intel (see [14]).

3.3.1 SPARC

A SPARC multiprocessor system has two special instructions to support tightly coupled multiprocessor systems: swap and atomic load and store unsigned byte (ldstub). Swap exchanges the contents from an IU (Integer Unit) register with a word from memory, while preventing other memory accesses from intervening on the memory or I/O bus.
The atomic load and store instruction reads a byte from memory into an IU register and then changes all the bits in the same byte to ones, assuring there will not be any intervening access to this byte by any other processors. It can be used to construct semaphores.
The SPARC specification defines that the MMU will have to make sure that changing the referenced or modified bit of a certain page table entry(PTE) in the TLB will result in also changing the PTE in main memory. This way the operating system can keep page tables correct.

3.3.2 Intel

When the system is started up, one processor is responsible for initializing the system and setting up the operating system. This processor is called the bootstrap processor (BSP). Other processors, applications processors (APs), remain idle during setup. The BSP is responsible for starting up the other processors when it has finished initialization. This is done by sending soft interrupts to the other processors.
The BSP can use the MP (multiprocessor) configuration table to find out which processors are available. For each processor there is an entry in the table. This table is written by hardware during setup.
The APICs (Advanced Programmable Interrupt Controllers) are collectively responsible for delivering interrupts from interrupt sources to interrupt destinations throughout the multiprocessor system. There can be several APICs in the system, which communicate with each other over a special bus. The APICs allow the system to scale better because of the following reasons:

- Off loading interrupt related traffic from the memory bus, making the memory bus more available for processor use.
- Helping processors share the interrupt processing load with other processors.

The local APIC units also provide interprocessor interrupts (IPIs), which allow any processor to interrupt any other processor or set of processors. There are several types of IPIs. Among them, the INIT IPI and the STARTUP IPI are specifically designed for system startup. Also IPIs exist for shutting down processors.
IPIs are used for handling a TLB or PTE invalidation as well. By sending a broadcast IPI, the operating system can make sure that each MMU invalidates its TLB or a specific PTE. Still memory based synchronization is needed to complete this action.

3.4 Software implementations

Next, an example of an SMP operating system will be discussed shortly. The example is Linux (see [15]).

3.4.1 Linux

The locking of kernel data can be done in several kind of symmetry[1] degrees. The first implementations of SMP Linux used a big lock around the kernel, which allowed only one processor to execute kernel code at the same time. Some assumptions are made to make sure that only one process can execute kernel code:

- No process running in kernel will be pre-empted by another kernel mode process unless it voluntary sleeps.
- Interrupts may pre-empt a process running in the kernel, but will always return to that process. A process may disable interrupts.
- An interrupt will not be pre-empted by a kernel task.

This is a pretty simple solution. The disadvantage is that the kernel can become a bottleneck, especially when the system has a lot of processors. Each time a processor wants access to the kernel it must wait until other processors have done their work inside the kernel, even though the processors do not need the same part of the operating system. Things can be made faster by using finer grained locks. This way processors can actually execute kernel code at the same time.

So, the first implementations of SMP Linux showed hardly any symmetry, because only one processor was allowed in the kernel. In partial symmetric kernels, more than one processor can enter the kernel, for as long as these processors execute within separate modules of the kernel. In true (up to a certain limit) symmetric kernels, processors are also allowed to execute within the same modules at the same time. The more symmetric the kernel is, the more synchronization tools are needed. These synchronization methods will result in some extra overhead for accessing and leaving the critical sections, but will keep processors from waiting to enter the kernel.

To change Linux from a uniprocessor operating system to an SMP operating system the following things had to be done:

- A single lock is maintained across all processors. Processors can enter the kernel spinning this lock.
- Change the boot up. The operating system must start up all the processors.
- A processor field is added to each task. The operating system must be able to see which processor is executing this task.
- Some changes in memory management were necessary. The operating system has to make sure data in all the TLBs stays correct.
- The scheduling algorithm hardly needed any change. Almost the same algorithm was used as in the uniprocessor system.

[1]Symmetry in this context has a different meaning than in SMP. In SMP it refers to the access a processor has to I/O devices and memory modules, while here it refers to concurrent execution of the kernel.

Later, the kernel was made more symmetric. Improving Linux for SMP systems still is a topic of research.

4 SMP OSP design

In this chapter the SMP OSP design is given. In it are the most important decisions made in designing the SMP OSP system. The main questions which arise are:

- How should multiple processors running in parallel be simulated?
- What functionality will the operating system have?
- How to do proper error checking?
- What should the simulated hardware look like?
- What locking mechanism should be used in the operating system?
- What scheduling and page replacement algorithms will be used?

All these problems will be dealt with further on. But before beginning to build the new design it is necessary to define a goal and describe what properties the design should show. Then the event engine, the simulated hardware and each of the operating system modules will be discussed. The last section of this chapter is devoted to error checking.

4.1 Goal

The main goal is to create a number of documents and files which can be used by the programmer(s) who want(s) to implement SMP OSP. After reading these documents, and reading the current OSP design[1], he or she should be able to understand how the current OSP program works and how it is designed and how to implement SMP OSP. SMP will not be a completely new design, it is very much based on the current OSP, which is a very useful system. Some changes, most of the times small ones, could improve the current system though. These changes will be made in the SMP design, they are explained in relevant sections of the new design.
Before making the design a few characteristics were defined which the SMP OSP design should have. These characteristics are not only important to the OSP programmer, but especially important to the student who will have to work with the program. (SMP) OSP is made for students in their first or second year to work with, so it is important to state goals which are most useful to the these student.
First, the system should be kept ***simple, but realistic***. Simple because the student should be able to learn how the operating system works and how to add code him or herself pretty quickly. Realistic because this will give the student the idea he or she is dealing with a real operating system. An operating system which is simple but not realistic might confuse the student.

[1]The uniprocessor design, the one already made, will be referred to as current OSP design, the new design, the one described in this section, as SMP (OSP) design

The system can me made more realistic by ensuring that all situations that may arise during simulation are theoretically possible. Most of the times only small changes are needed in the current OSP to avoid these unrealistic simulations.
SMP OSP will be more difficult to work with then the current OSP since especially the synchronization of all object will make the system more complex. But still, it is tried to keep SMP OSP from getting too complex, so it can be used at the same level of education as the current OSP.
A second goal is to keep the design ***modular***, which is an issue in each design. On the one hand this means that the hardware, operating system and event engine modules will be separated and only a small interface between these modules should be allowed. On the other hand, the modules within the operating system will also be separated. In the current OSP there are some responsibilities spread over two or more modules, which actually belong together. To give an example, the swap file is opened in the page fault handler, but closed in the disk interrupt handler. If the swap file is opened and closed in the page fault handler, it becomes much easier to understand the program. Sometimes is hard to avoid this type of problem, but most of the times a small change is enough.
The advantage here is that a modular design is easier to understand. Clearly defined modules with a small interface make it easier to get a good overview of the whole, complex system. Secondly, a modular design is easier to change. Chances are smaller that an update or bug fix will result in changing several modules.
A third advantage of a modular design in OSP, is that each assignment is less dependent of other assignments. Students will have to spend less time in finding out what responsibilities the other modules have and what he or she should implement. This way it is easier for teachers to skip one or more assignment and students who miss an assignment will have less trouble in catching up.
A third goal is to build a ***transparent*** design, especially transparent to the student. To give another example from the current program, the student does not know how the IFL module is designed. The IflIORB class is a subclass of the event class, but this is not visible to the student. He or she has to know about events though. Of course the teacher can tell this to the student, but it would be less confusing if the student would be able to see how the complete operating system is designed.
Another issue to keep in mind is how much code from the current program can be used in the new implementation. By trying to keep most of the methods more or less the same, the programmer can reuse a lot of code from the current OSP. Sometimes a balance has to be found between using old code and improving the design. In some cases designing SMP will look more like changing the current OSP than like making a new design. Especially in the IFL and event engine modules, which are not visible to the student, changes are tried to be kept to a minimum. Maybe the term redesign is more appropriate.
Aside from the things described here, also the suggestions for improving the current OSP will be used in designing SMP OSP. The main changes will be:

- As stated before, keep a clear distinction between the operating system, the hardware and the event engine by defining a small interface between them. For example, define two separate events, one used in the event engine, and one in the operating system.
- Create a more object oriented design using fewer static classes and methods.

Maybe working with OSP is the first time the student has to program in an object oriented language and deal with an object oriented design. So it is important to use the mechanisms used in object oriented design and language properly. To give an example, not all classes will be a subclass of GlobalVariables.

- Use meaningful names for attributes and methods, referring to the responsibilities or use.

In the process of designing SMP OSP, the module which is responsible for doing the error checking will be designed after the modules the student should implement. The reason is a more realistic design of the operating system is created this way. In the design of the operating system, a lot of thought has to be given to whether or not all situations are realistic and to the modular design of the system. This will result in a clear design of a realistic system.
Only after the new operating system design is finished, the IFL module will be designed. In the design of the operating system, the thought that later the IFL module will be added has to be kept in mind. The most important questions left after designing the operating system are: what will be the general way of doing the error checking and what should be checked in each module? This will result in moving some of the attributes from the students module to the IFL module, but at least the operating system will be clear and realistic. More about the error checking can be read in section 4.5.

Documents

The way of describing the design will be more or less the same as the design of the current OSP. This chapter outlines a global design, though for some classes and methods a detailed description is given. Especially to those parts of the design which are different from the current OSP design more attention is given. Details about every method and attribute can be found in the Java class files, which can be used by the OSP programmer. The following documents/files will be produced.

- For every module a description about what is added or changed in this module.
- For every module a class diagram.
- A collaboration diagram of the operating system.
- A number of Java files containing all class definitions, methods and attributes. From these files HTML pages can be created which contain descriptions about all the classes, fields and methods by the Java doc generator.

So, this document gives an overview of the system, and describes the choices made in designing SMP OSP. The Java files function as a starting point for the programmer(s) who will implement SMP OSP.

4.1.1 Simplifications

As stated before, the system should be simple but realistic. To establish this some simplifications have to be made, just like in the current OSP. The chosen simplifications are pretty much the same as in the current OSP. The most important ones are:

- no data in memory, files or devices.
- the size of a page is the same as the block size of all devices.
- no interrupts during the handling of a system call and no nested interrupts.

Other simplifications will be discussed in each of the sections about the hardware module and the modules in the operating system. Concerning the interrupts, the reason for not allowing interrupts while a processor is in kernel mode, is that things would become too complex otherwise. The event engine needs a lot of change to simulate this. Suppose an interrupt is scheduled on a certain processor, but the processor is already running in kernel mode, thus a thread is executing a certain system call. This thread should then be stopped and a new thread should be created and start the interrupt. These kind of interrupts also make it hard to do error checking.
Not only the event engine would get more complex, the operating system as well. The student has to use interrupt masking, since he or she can not rely on the fact the thread will always run until it goes to sleep or return voluntarily. It will still be possible though that one CPU receives an interrupt while the other is in kernel mode. This means that the interrupt handlers also need to use the synchronization methods.
Though most real operating systems probably allow recursive interrupts, it would certainly not be fatal to not allow them, except for real time system, where the time to respond to certain interrupts has to be kept to a minimum. Thus allowing this simplification is still results in realistic system: the operating system from OSP is not made for real time systems.
A final remark concerning simplifications: the operating system will not provide fault tolerance and graceful degradation. The design of the simulated hardware is kept simple as well, as can be read in section 4.3.

4.2 Simulation and event engine

4.2.1 Simulating parallel running processors

A main problem which arises in designing SMP OSP is how to simulate the execution of more than one processor. In the current OSP, threads cannot be preempted. This means that a thread will continue executing kernel code until it voluntarily leaves the kernel or until it calls suspend(). To have the suspending of a thread look realistic to the student, in the current OSP program Java threads are used. The Java thread will stop executing the code when an OSP thread suspends itself and the engine thread will start issuing a new event. This results in a realistic system.
In SMP OSP, the same way of simulating could be applied. In that case, one big lock around the kernel would be simulated. Only one thread at a time is allowed to execute in the kernel. A lot of code from the old system can be used then. The only thing that has to be changed is the simulated hardware, the operating system can be left more or less the same. Although some real systems use this method of dealing with multiple processors, in most of the cases a more symmetric kernel is implemented.
SMP OSP can be made more interesting by creating a more granular operating system. Two possible ways of simulating this kind of parallel kernel execution can be distinguished. The first is running several Java threads concurrently.

This way kernel code will be executed by one or more threads at the same time without synchronizing these threads in the IFL module, which is done in the current OSP. The student has to use locking mechanisms to protect all the data structures. The advantage here is that the student is forced to use the synchronization methods carefully, since the Java thread package schedules the Java threads.
This is not the case with the second option, which is providing the student with some simulated synchronization methods. These are not Java synchronization methods, but the student has to use them because the IFL module will check this. When they are not used properly, an error will be returned. This is kind of simulation is used in the current OSP.
The second method is easier to implement. No Java threads will run in concurrently so no changes in all the modules will be needed in order to make them resistant against concurrently running threads. Also, it is much easier to give proper feedback. The IFL can check whether the locks are acquired properly and return specific error messages in the case they are not correctly acquired. Thirdly, the system will stay deterministic. When using real threads it is very well possible the system will not give the same output when executed twice with the same parameters, though this depends on the system on which OSP is run and whether or not other applications are run at the same time.
But there is one major argument against the use of simulated locks. They are not realistic. The student will not get the feeling several processors are running in parallel, simply because there is only one Java thread running. There is hardly any difference with the current OSP.
The student can check the log file which contains all kind of information about the execution of the system. The log file will show that each system call runs to an end before another system call or interrupt occurs. There is no interleaving between the output of several system calls.
This argument resulted in choosing for the use of Java threads. Along with this choice some problems arise about how to synchronize data structures in all parts of OSP. How these are solved is discussed throughout the rest of this document.

Another problem is the scheduling of the Java threads. To check the code made by the student, one would like to simulate certain executions paths, since certain problems only will become clear in very specific situations. For example, deadlock can only occur when two threads have locked certain data structures and at the same time want to acquire each others lock. Chances are not very high such a situation will occur in the simulation.
The problem is that the simulator does not know what the students code looks like and thus these specific situation can not be simulated exactly as one would like to. There is way to have some influence the simulation though. Each time a method is called from the students module, the IFL module takes over control, since all methods have to be called via the IFL module. This is the place where the OSP programmer can influence the Java thread scheduler by for example yielding other threads to run first. So in the case of synchronization, the IFL module can make sure that a thread acquiring a lock, has to wait until another thread also acquires the lock. If there is a way in which deadlock can occur because of an error in the students code, chances can be made higher this will occur.
Another way to check the synchronization methods (especially the error check-

ing of the synchronization methods has to deal with this problem, other error checking can be done the same way as in the current OSP program) is to increase the number of system calls which influence the same data structures. For example, the event engine can make sure that one thread will create a new thread for a certain task, while another will kill that same task. The system calls both have to change the task data structure and also the new thread data structure, so synchronization is very important here.
These problems would have occurred in a single thread execution as well. The only way to increase the chance of certain errors occurring is to define certain execution paths and try to execute these paths in the simulation by influencing the Java thread scheduler using provided methods from the standard Java thread class.

4.2.2 Event engine

Since the event engine works fine in the current OSP, it is tried to keep the event engine the same as much as possible so a lot of code from it can be copied into the SMP OSP system. Only some changes are needed concerning the use of multiple processors. Some small improvements are made as well which result in an event engine which is much easier to understand. These improvements are the ones discussed in section 2.2. They do not change the functionality of the event engine.
The event engine consists of an engine thread, an event queue and an event tree. How these objects should be changed in order to simulate multiple processors is discussed next.

Engine thread

In the current OSP one engine thread is used. This thread runs in a constant loop in which it removes an event from the event queue, activates it and evaluates it, putting a new event on the event queue. Since only one OSP thread is allowed in the kernel, using one Java thread is sufficient.
One problem the OSP programmers encountered, was how to deal with a thread that goes to sleep while operating in kernel mode. This was solved by creating a new Java thread for those system calls or interrupts which could result in suspending the current OSP thread. When the OSP thread gets suspended, the Java thread puts itself to sleep and wakes up the engine thread. Suspending and resuming the Java threads is the responsibility of the IFLmodule. The student should be unaware of this fact, except for noticing that OSP threads indeed are suspended and resumed again later, by looking at the output of the system.
In SMP OSP the same mechanism can be used, only some things need to be changed. For each processor defined in the simulated hardware, one engine thread is created. Each engine thread will do the same things as in the current OSP, i.e. remove a thread from the event queue, start the event, create a new event and put the new event on the event queue. As in the current OSP, in some cases a new Java thread is created to execute the system call or interrupt handler. When this thread goes to sleep, simulating an OSP thread suspending itself, the associated engine thread takes over again and starts a new event.
Since a system call and a page fault should only occur while an OSP thread is running on the simulated processor, each time one of these events comes up, a

check is needed to see if there is an OSP thread running on the processor. If not, the event is discarded.
Another solution for simulating several processors one could think of is that each OSP thread has a Java thread attached to it. Each time an OSP thread gets dispatched, the Java thread is resumed. This thread removes an event from the event queue and activates it. This solution would need more changes in the event engine code though and it has no advantages opposed to the chosen solution.

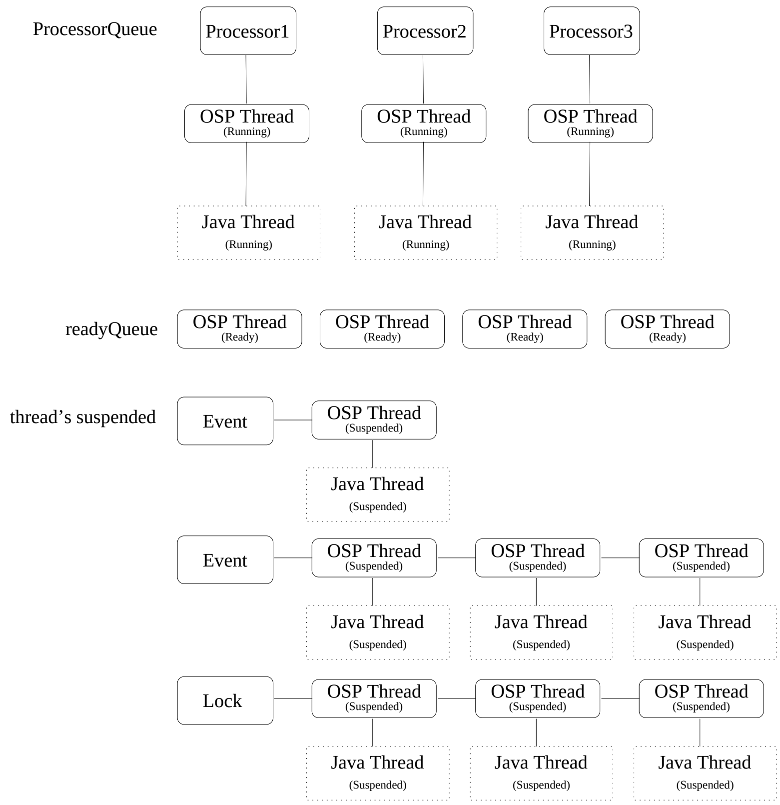

Figure 4.1: *Snapshot of the simulation: Java threads and OSP threads*

So, for each processor an engine thread is created. These engine threads can not be killed during simulation. For each engine thread there will be a separate lock, which is used in the same way as in the current OSP (see 2.2.6).

Events

In the current OSP there is no distinction between an operating system event and an event engine event. The first event is some kind of action a thread can wait for. The second is an event which is placed in the event queue and defines which system call or interrupt should be simulated by the event engine. This is very confusing, not only to the student, who in confronted with objects from the event engine, but for the OSP programmer as well.
To avoid this kind of confusion, two separate kind of events are defined in the

SMP OSP. The event containing the reference to the callback object which generates the system call or interrupt is part of the event engine, and is something the student is not supposed to know about. That is, he or she should not be confronted with it when implementing one of the system calls.
The other event class, which is part of the operating system, represents an event one or more threads can be suspend on. More about this type of event will be discussed in section 4.4.1.
Within the event engine two types of events can be distinguished. The first type of event does not require a specific CPU to be executed on. For example, a send() system call which sends a message to a port, could be issued by a thread on any CPU. For the second type of event, it does matters on which CPU the event is executed. Only a limited number of this type of event exists. These are the timer and interprocessor interrupt and the event which resumes a Java thread(not visible to the student). A disk interrupt will first be handled by the interrupt controller. This can be done by any thread. But the IC will then choose a CPU to send the interrupt to. This interrupt is CPU-specific. Thus actually, a disk interrupt is CPU specific.
By adding a field to the event class (from the event engine), which defines on which processor the event should be simulated, the event can be activated by the engine thread associated with that processor.

Event queue and event tree

In the current event engine the most important objects are the engine thread, the event tree and the event queue. In SMP OSP, several engine threads are created, but what about the event queue and event tree? Though these are mainly implementation issues, one important question to answer before starting the implementation is how to spread the events created by the event tree among the processors. It would be best to find a way which needs a minimum of changes to the event engine code.
A solution would be to create a new event tree for every processor, but then each processor would handle exactly the same system calls in exactly the same order. So a better way is to spread the events created by the event tree randomly among the processors. Only one event tree is needed, just like in the current OSP. At the beginning an evaluate message is sent through the tree, after which the event queue will contain the first events.
The only problem which is left then is how to keep track of time. In the current OSP each event is associated with a time stamp. The events in the event queue are ordered by this time stamp. In the log file before printing a message, the time associated with the event is printed.
In SMP OSP the time field can still be used to order the events in the event queue. But printing the time in front of each message in the log file, would result in strange time loops. For example, suppose processor A starts an event at time 15. The next event is scheduled at time 17. Processor A starts executing the event, also printing messages to the log file about what it is doing. Then the Java thread scheduler decides to run the Java thread (engine thread) which is associated with the OSP thread running on processor B. It starts executing the next event, scheduled at time 17. Messages will also be printed to the log file, now beginning with '17:'. After a while the Java thread associated with processor A will run again. Now messages beginning with '15:' will show up in

the log file.
Instead of printing the time associated with the event, a global time value needs to be printed. Each time an event is taken from the event queue, this global value is updated to the time stamp associated with the event. The exception is when a CPU specific event comes up. This event can only be run by one specific engine thread, so other engine threads might already start the next event. In that case the global time value will be higher than the time stamp of the event, thus it does not have to be increased.
No changes are needed in the event tree, only the event tree might be build up differently. This is a decision which should be made for each operating system module during implementation and has nothing to do with the code which implements the functionality of the event tree.

4.3 Hardware

A simple hardware system will be simulated in SMP OSP. The hardware consists of a single memory module, a number of CPUs (the number can be adjusted, using the parameter file) and a number of I/O devices, like in the current OSP system. The memory is a UMA (Uniform Memory Access) architecture. Each CPU has a MMU (Memory Management Unit), which translates virtual addresses into physical addresses. There is no translation lookaside buffer (TLB) and no cache memory. Figure 4.2 shows what the simulated hardware looks like.

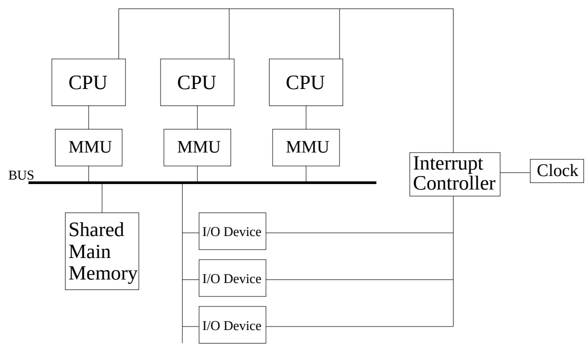

Figure 4.2: *The architecture of the simulated machine in SMP OSP*

Not like in the current OSP, this hardware provides interprocessor communication. This means that a processor can send an interprocessor interrupt (IPI) to another processor. There will be only one kind of IPI. Using variables in the operating system, CPUs can provide each other with more specific information about the IPI.
There is only one clock in the whole system. The only functionality a clock has, is producing clock interrupts. An interrupt controller (IC) is added as well. This IC can be used to send interrupts to a specific CPU. The operating system can set the next CPU to be interrupted. This way, interrupts can be sent to a CPU which is idle.
The most important change in the machine design is that the hardware module is separated from the operating system. The only connections the hardware

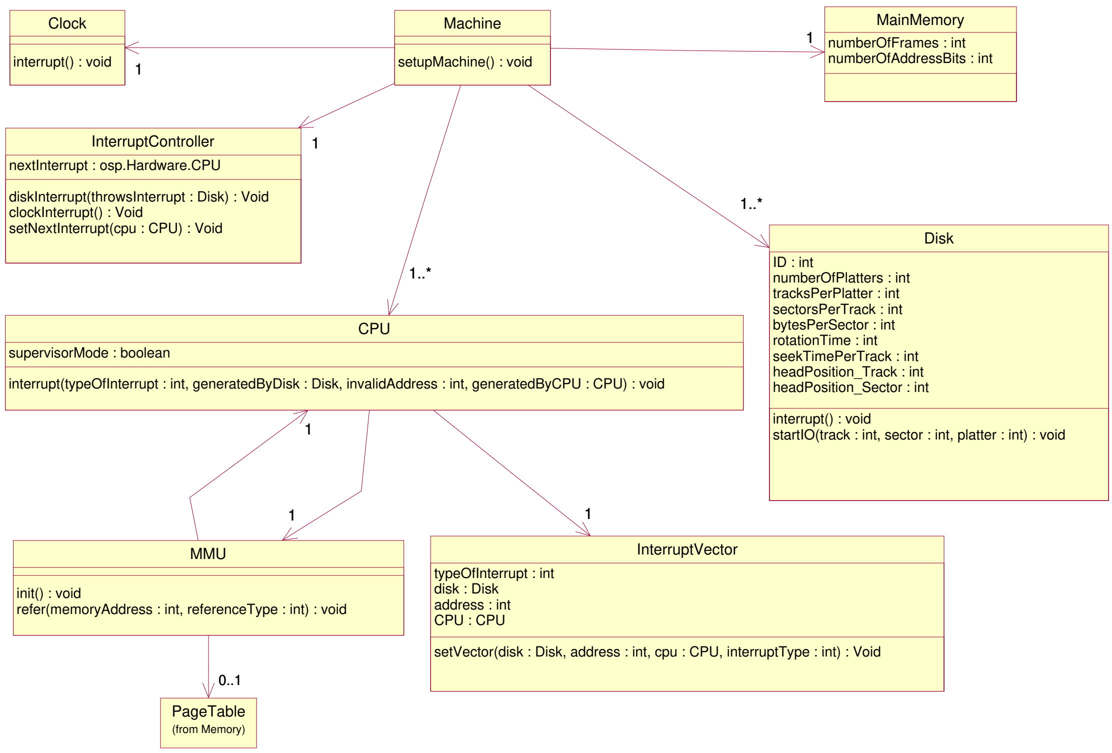

Figure 4.3: *Class diagram of the hardware module.*

has to the operating system are the page table and the interrupt vector. In a real system, these are stored in main memory. The MMU holds a pointer to the page table of the process currently running. This pointer can be set by the operating system. The interrupt vector is normally stored at a fixed place in memory. In most real systems each device will be assigned a separate piece of memory to store information about an interrupt. But in the SMP OSP simulation, main memory is actually empty. The only things which are stored in class MainMemory are the number of frames, and the number of address bits. One could argue that the interrupt vector should be in main memory. But then, so should the whole operating system. This is not possible, since the hardware is simulated, while the operating system is in a way real. So the interrupt vector is stored in the operating system, as well as each page table. Each CPU has its own interrupt vector and each MMU has a reference to a page table.
This is a simple architecture to simulate. The following functionality could be added to make it more complex:

- Add an extra field to a CPU which defines the clock speed of this CPU. This way, the operating system has to keep the fastest CPUs from being idle as much as possible.
- Instead of using UMA, a NUMA architecture can be simulated. The operating system then has to determine what is the best place for a page to be

stored.

- Add cache memory to each CPU. The student then has to deal with the cache consistency problem, even though this is implemented in hardware, just like the translation of virtual addresses in the MMU.
- Add a TLB (Translation Lookaside Buffer) to each MMU. The operating has to flush (part of) the TLB when pages are swapped in and out.

This kind of functionalities is not included in this design.

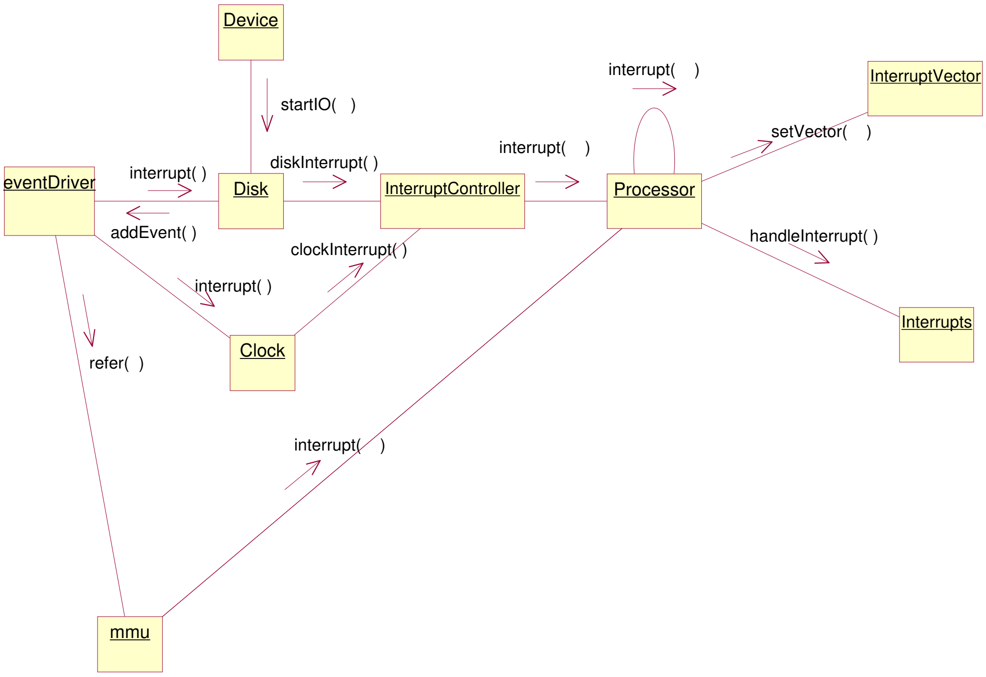

Figure 4.4: *Collaboration diagram of the hardware module.*

4.3.1 Interrupts

The major functionality this hardware shows is handling interrupts, passing them to the operating system. Four kind of interrupts can occur in this system:

- Timer interrupt produced by the clock.
- Page fault interrupt produced by a MMU.
- Disk interrupt.
- Interprocessor interrupt.

How interrupts are handled is shown in the collaboration diagram (figure 4.4). Each interrupt flows through the system and reaches a CPU. The CPU sets up the interrupt vector and calls handleInterrupt() from class Interrupts. This class is part of the operating system. So this is the place where CPU will jump to supervisor mode and the kernel takes over. It is up to the operating system to deal with the interrupt.

Timer interrupt A clock interrupt is generated periodically. The event engine sends an interrupt message to the clock. The clock sends a message to the interrupt controller. It is up to the interrupt controller to direct the interrupt to one of the CPUs. Which CPU to interrupt can be set by the operating system using the method setNextInterrupt(). Finally the CPU calls the general interrupt handler.

Page fault interrupt A page fault interrupt is generated by the memory management unit (MMU). The event engine calls refer() a-periodically. The MMU will check if the give virtual address is correct. If not, an interrupt will be generated. The interrupt message is sent to the CPU directly, because each CPU has its own MMU.

Disk interrupt A disk interrupt can only occur after a read or write operation is issued to the disk. This is done by the operating system by calling startIO(). The disk will calculate how much time it takes to move its head to the correct track and sector. Then, it will add an event to the event queue, specifying at what time in future the disk interrupt should occur. An engine thread takes the event from the event queue at the specified time and sends the interrupt message to the disk. The disk then calls the IC, which directs the interrupt to one of the CPUs.

Interprocessor interrupt An interprocessor interrupt (IPI) can be used to send a message from one CPU to another. For example, when a task is killed, all threads from that task need to be removed from the CPUs. The thread handling the system call, killing the task, can then send an IPI to those CPUs which run a thread that needs to be killed. When a CPU receives an IPI, it will call the general interrupt handler. By checking certain variables inside the operating system the interrupt handler can check why it was interrupted (see 4.4.4).

So the major changes are that interrupts are started in the object which creates them. This is more realistic way of simulating then, like in the current OSP, by calling the dispatch() method immediately when a timer interrupt is simulated, without intervention of the hardware.
One thing to keep in mind is that a page fault interrupt can only occur when a thread is running. The other three interrupts from the list can occur at any time. The page fault handler is the only interrupt handler which can be suspended. The other interrupts can also occur when no thread is running, so they can not be suspended.

4.3.2 Classes

The classes from the hardware package are much alike with the one from the current OSP. For each class a description will be given, explaining the purpose of this class.

Machine The machine holds static references to all objects which are part of the hardware. One can access these objects using general get methods defined in this class. These methods are not shown in the class diagram. By calling setupMachine() all parts of the hardware are initialized.

Clock The only responsibility a clock has, is creating interrupts on a regular basis. These interrupts are started from the event engine. The event engine calls the interrupt() method from the clock.

InterruptController Clock and disk interrupts are sent to the interrupt controller. It has the responsibility to send the interrupt message to one of the CPUs. The operating system can set the CPU which should be interrupted. The programmer can choose to only set the CPU once, at the beginning of the simulation, or to constantly change this processor. This can result in better performance, interrupts can be sent to CPUs which are idle.

CPU A CPU makes sure that the general interrupt handler is called each time an interrupt message is sent to the CPU. Before calling the interrupt handler it sets up the interrupt vector. Each CPU has its own interrupt vector.
A CPU also has one attribute which defines the state of the CPU. If the CPU is in supervisor mode, it is executing kernel code. This should be set to true when the CPU jumps to the operating system, for instance when the CPU calls the general interrupt handler.

Disk In the current OSP a disk is a subclass of a device. In order to keep the distinction between hardware and operating system clear, for each disk in hardware a separate device is defined. Each device holds a reference to exactly one disk. This means some changes are needed in class Disk. A disk is part of the hardware, thus it can not handle IORBs (I/O Request Blocks). When a disk read or write is issued, instead of sending an IORB to the disk, a new method, startIO(), in this class is called to start the disk operation.
A disk contains a number of values describing the characteristics of this disk. A disk consists of a number of platters. Each platter has a fixed number of tracks, each track has a fixed number of sectors, each sector has a fixed number of bytes.
Two other characteristics are seekTimePerTrack, the time it takes for the head to move from one track to its neighbor track, and rotationTime, the time it takes a platter to rotate exactly once.
When the startIO(track, sector) method is called the disk sends a message to the event engine to add an interrupt event to the event queue. After a while, the event engine will call interrupt() from the disk. The disk updates the position of the head and sends the interrupt message to the interrupt controller.

MainMemory Main memory does contain date in SMP OSP. Only the number of frames and number of address bits are defined. This second value defines how many bits are used to specify a certain address within a frame. From these two values the total length of main memory can be calculated. To keep things simple, a UMA architecture is used, so no distinction needs to be made between several parts of memory.

MMU The memory management unit is responsible for translating virtual addresses into physical addresses. The event engine calls refer(address, referenceType) once in a while. The MMU will search through the page table for the correct frame. If the frame is not stored in main memory, an interrupt will be generated.
Just like in the current OSP, class MMU is one of the parts which can be implemented by the student even though the MMU is part of the hardware.

It is useful for the student to know how virtual addresses are translated. The init() method is added so students can setup static variables.

4.4 Operating system

In the next sections the design for each of the modules forming the operating system is given. These are: Locks, Memory, FileSys, Devices, Tasks, Threads and Ports. Most attention will be given to the synchronization of kernel objects and changes from the current OSP. A collaboration diagram of the operating system can be found in appendix B.4.
Before focussing on each of the modules first the class Event will be discussed, because this class is used through the whole system. Event is part of the threads module.
For reasons explained before, in the discussion of the modules, no distinction will be made between the IFL module and the students module. Only in section 4.5 attention will be given to the error checking.

4.4.1 Operating system event

When a thread suspends itself, it must specify what it is waiting for. Therefore it must specify an event, upon occurrence of which the thread can be resumed again. An event in SMP OSP contains a list of threads. These are the threads waiting for the event to happen. Since events are so much involved with OSP threads, Event is part of the threads module. Its class diagram can be found in figure 4.7. In SMP OSP three kind of events are defined:

- *diskEvent*: data needs to be moved to or from a disk.
- *pageFaultEvent*: a page which is not in main memory, is swapped in.
- *portEvent*: a message is added or removed from the port's buffer.

For each of these events a subclass is defined. When the event has occurred, the notifyAll() or notifySingle() method is called. In the first case all threads are resumed, in the second only one is resumed (in a first in first out order). These methods are defined in the event class. Another method is threadKilled(), which should be overloaded in each of the subclasses. This is called when a thread, waiting for the event, is killed. For each of the events different actions need to be taken. Maybe, if no other thread is waiting, the event should be cancelled. This is the case with a disk event. A more detailed specification of the subclasses can be found in the other section dealing with the memory, device and port modules.

```
wait():
        add the current thread to the waiting list
        suspend the thread on this event

notifyAll():
        (the event has occurred)
        resume all threads waiting for the event
        remove all threads from the list
```

```
notifySingle():
        (the event has occurred)
        resume a single thread waiting for the event
        remove the thread from the list

threadKilled(thread)
        remove the thread from the list
        (different for each subclass)
```

To protect an event against multiple processors accessing the list of waiting threads at the same time a spin lock is used (more about locks can be read in section 4.4.2).

An important change with the current OSP is that instead of making IORB and Port a subclass of the Event class, an extra attribute is added to the IORB and Port class. The advantage is that the student has better insights in the structure of the system. In the current OSP, IflIORB is a subclass of Event. IORB is a subclass of IflIORB, of which the definition is not visible to the student. This change results in more transparent design, which is one of the goals stated at the beginning of this chapter.
A second reason for not keeping IORB and Port a subclass of Event is that now the structure is more intuitive, from an object oriented point of view. A port is not an event, so a port should not be a subclass of an event. An IORB is no event either. It is a request block, a data structure with information about an I/O operation. Thus another goal of the design is satisfied for this part of the operating system. PageFaultEvent already is an attribute of a page in the current OSP.
Within SMP OSP, the difference between an event and a lock does not seem that big. Both classes block and resume threads. But the difference is that, at a certain level of abstraction, an event is an object waiting for something to happen and a lock is used to protect objects or data structures against concurrent access.

4.4.2 Locks

One of the main questions which arises in designing the operating system is how symmetric it should be, i.e. at what level are the operating system data structures protected. One option would be to use one big lock to protect to whole kernel, but as mentioned before, this is not very realistic and will not result an in interesting new SMP system. A multiprocessor system should try to keep the processors running as much as possible. This way the performance of the system will go up. So another option is to use smaller locks. This is more realistic, but also a more complex to implement. Synchronization methods will have to be used in order to keep kernel data structures correct.
In SMP OSP, an extra module is added, which provides the locking mechanisms that can be used in the rest of the operating system. This module is defined in the same way as the other modules. An extra assignment can be defined for which the student has to implement the methods defined in the locks module.

This is very useful since locking mechanisms and issues involved with locking, like deadlock and starvation, have always been a difficult topic for many students. An assignment which includes implementing several kind of locks can be useful to these students.
In this new module several locks have to be implemented. These are the following:

- *spin lock*: the thread tries to acquire the lock over and over again, without releasing the processor.
- *semaphore*: the lock is passed on to one thread when release() is called.
- *mutex lock*: all waiting threads are resumed when release() is called.
- *read/write lock*: several reading threads are allowed, but only one writing thread.

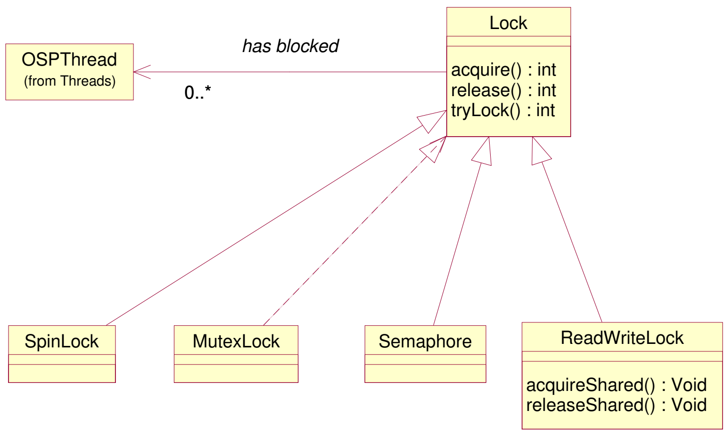

Figure 4.5: *Locks class definition*

These 4 locks are subclasses of class Lock as can be see in figure 4.5. Each class overloads the methods acquire(), release() and tryLock(). For the last three locks from the list the responsibilities for these methods are described in the following pseudo code:

```
acquire():
        while (lock not acquired)
                try to acquire the lock
                if the lock is already taken
                    suspend the current thread

release():
        (only semaphore)
        pass lock to next thread
        resume the thread
```

```
release():
        (only read/write and mutex lock)
        release the lock
        resume one or all threads blocked on this lock

tryLock():
        try to acquire the lock
        if lock acquired
                return success
        else
                return error
```

The spin lock is different from the other three locks, because a thread trying to acquire a spin lock cannot be suspended. It tries to acquire the lock over and over again. This results in the following pseudo code:

```
acquire():
        while (lock not acquired)
                try to acquire the lock

release():
        release the lock

tryLock():
        try to acquire the lock
        if lock acquired
                return success
        else
                return error
```

The read/write lock uses two extra methods, which can be used to specify whether the data structure will be changed or not. The methods acquire() and release() are used to specify that the thread wants to write. The method acquireShared() can be used to specify that several threads are allowed inside the lock. The method releaseShared() is used to release the lock again. How to implement such is a lock is up to the student (see [4] for examples).

Usage

A thread is allowed to possess several locks, while going to sleep. To remember which locks the thread owns, an extra attribute was added to the thread class. When a thread is killed, it should release the locks, so other threads can get access to them. If a thread is killed while it is blocked on a lock, the thread is not removed from the lock's waiting threads. This is not necessary, as long as is made sure that a lock is not passed to a dead thread. The only lock for which this can happen is the semaphore. So when releasing a semaphore, it has to made sure that the thread to which the lock is given is not dead. This can be done be reading the status of the thread. For more information about (killing) threads, see section 4.4.4.
In order to protect the non-spinning locks (semaphore, mutex and read/write), a spin lock is needed. In a real system the spin lock can be implemented using

one of the operations provided by the hardware, for example an atomic read and write operation. In SMP OSP, this can be done by using one of the Java synchronize methods.
When a thread tries to acquire a non-spinning lock, which is already locked, the thread gets suspended. To suspend a thread, an event has to be specified. It would be useless to create an event for every lock, because then both the lock and the event will hold references to the blocked threads. So, in this case the lock will call suspend(null). The lock on which the thread is blocked does not need to be specified. When a thread is blocked on an event though, the event has to be specified because when a thread is killed, maybe the event has to be cancelled. For locks this is not the case.

Use of the locking mechanisms in the rest of the system

All the objects in the operating system need to be protected against concurrent access. The question is how many locks should be used. It is not very interesting to use one big lock around the whole kernel, therefore, most objects will have their own lock. In a real system a problem could be that all the locks take too much space in memory. In OSP this is not a problem, the number of objects is limited.

object	**lock**
Thread	spin lock
Event	spin lock
readyQueue	spin lock
processorQueue	spin lock
Task	spin lock
Port	semaphore
Device	spin lock
PageTableEntry structure	spin lock
PageTableEntry data	read/write lock
all frames	spin lock
FileSys	mutex lock
OpenFile	spin lock

Table 4.1: *Locking mechanisms for the most important data structures in the operating system of OSP.*

In most cases a spin lock is used to protect a data structure. The reason for this is that most of the time only a single attribute needs to be read or written. This does not take much time, so a spin lock is the most logical. PageTableEntry uses two locks, why is explained in section 4.4.6.
Three exceptions can be noticed, ports, the file system and the page table entry. In the case of the ports module, there are two reasons for choosing a semaphore. First, sending to or receiving from a port can take some time. Data needs to be moved from or into the buffer. In OSP no actual data is moved, but one could imagine, in a real operating system, this can take some time.
The second reason is that messages should be sent in a first in first out manner. So the thread which tries to send a message first, should be the one which puts

a message in the buffer first. When a mutex or spin lock was used, this would not be the case.
The file system is protected by a mutex lock since system calls to the file system most of the times imply that several data structures, like OpenFile and DirEntry need to be updated. So a spin lock could keep processors busy to long.
Of course, another reason for using a semaphore and mutex lock here is to make sure these locks are used somewhere in the operating system. It would be less interesting to the student to implement a lock which is not used, thus not tested. But it has to be noticed, the use of these locks here is not forced, these choices can be defended very well. This is also the case with the use of a read/write lock to protect each page table entry. More about this can be read in the part about the memory module, section 4.4.6. In each of the sections about the operating system modules, it is specified how the locks should be used.
The student has to know which locks are used to protect all the data structures inside the operating system. Otherwise he or she might have some trouble preventing deadlocks to occur. This will be explained next.

Deadlock prevention

Since locks are used through the hole system, deadlock is something to take into account in all parts of the operating system. In each module the student is confronted with the problem of deadlock.
In order to provide a reliable operating system, deadlock must be prevented. The student must make sure deadlock cannot occur, following the guidelines described in the next paragraphs. One must realize, deadlock avoidance and deadlock detection, methods in which no guarantee is given that deadlock will not occur, are used to protect data structures which can be locked by user programs. In that case, system calls are provided to lock these data structures and deadlock avoidance or detection methods can be used to prevent the system from crashing. At the moment, OSP does not provide these system calls.
What needs to be prevented exactly? Problems arise when a thread (thread A) wants to acquire a lock (lock 2) while already holding a lock (lock 1). Thread B, already holding lock 2, might want to acquire lock 1, which results in a deadlock. This is called a circular wait.
In some cases though, acquiring a lock, can never result in a deadlock though. These are locks which are acquired only to read or set certain values from the protected object. While holding such a lock, no other locks will be acquired, so no circular wait can occur. Acquiring a lock from one of the following objects can not result in a deadlock:

- readyQueue
- processorQueue
- Event
- OpenFile
- Port

It is up to the student to make sure that these locks will not cause a deadlock. In many cases this can be established by releasing a lock before acquiring another. In some cases the tryLock() method can be a solution.
In many cases status fields can help to keep the time holding a lock short. For example, when killing a task, first the task's lock needs to be acquired. Then

the status field of the task is set to 'killed' after which the lock can be released again. Next, all threads and ports created by this task can be killed without keeping the task's lock. Other methods should always first check the status field before accessing one of the attributes from a task. If the status field is set to 'killed' there is no use in changing the attributes. Since all attributes are accessed via the class methods, these are the only methods which have to check the status field. The student must keep in mind that a thread, port or task can always be killed by another processor while accessing these objects.
In other cases, to prevent deadlocks from happening, an order is defined in which locks need to be acquired. This is called hierarchical locking. The order is the following:

1. task
2. thread
3. file system
4. device
5. frame table entries
6. page table entry structure
7. page table entry data

The reason for choosing this order is based on the expected way of accessing objects. For example, most of the times first a task is accessed before a thread. A same kind of reasoning can be given for the rest of the list. During implementation this order can be changed if it turns out that this one is not efficient.

Besides deadlock, starvation could also occur. For example, when a mutex lock is used, it is possible a thread tries to acquire a lock over and over again, since all blocked threads are released again. The spin lock shows the same problem, the semaphore not. A semaphore always passes a lock to another thread in a first in first out manner. In OSP starvation is something which does not have to be prevented.

4.4.3 Tasks

A task in SMP OSP is mainly a collection of resources. It holds threads, ports and a page table. This is the same in the current OSP except for the fact that there is no reference to the swap file anymore. When a page needs to be swapped, the swap file must be opened by the thread causing the page fault. The open file is added to the list of open files in the thread. Thus, all open files are kept inside the thread, including the swap file. The swap file still needs to be created when a task is created though. The name of the swap file is always the same as the id the task owning the swap file.
Another difference in the issuing of system calls. A system call can only be done when a thread is running on the processor. So the event engine (or actually, the

callback function) needs to make sure that a thread is running when it simulates a system call.
In order to start the ball rolling, the OSP simulation starts with the creation of a task. This is the init task. A thread will be created for this task. This way it is realistic to simulate system calls. It would not be if system calls were generated while no thread is running, which is possible in the current OSP.

Synchronization As for synchronization concerned, each instance of Task has a spin lock, which is used each time one of the attributes is accessed, whether reading or writing. A spin lock is chosen here, because most of the times, only one of the attributes has to be read or set, which does not take much time.
The status field is important here, since when a task is killed, first this field has to be changed. When other threads try to access the task, they will notice that the task was killed and will not use access the task anymore. More about this way of synchronization can be found in the section 4.4.2.
A final small change with the current OSP, is that the page table is not of a fixed length anymore. When a task is created, the amount of memory (number of pages) needs to be given, so the size of the swap file can be determined. It is possible a thread tries to access a memory location outside the tasks page table. In that case the page fault handler should return an error.
Next, the responsibilities of the most important methods will be given in pseudo code. The acquiring and releasing of the spin lock is left out.

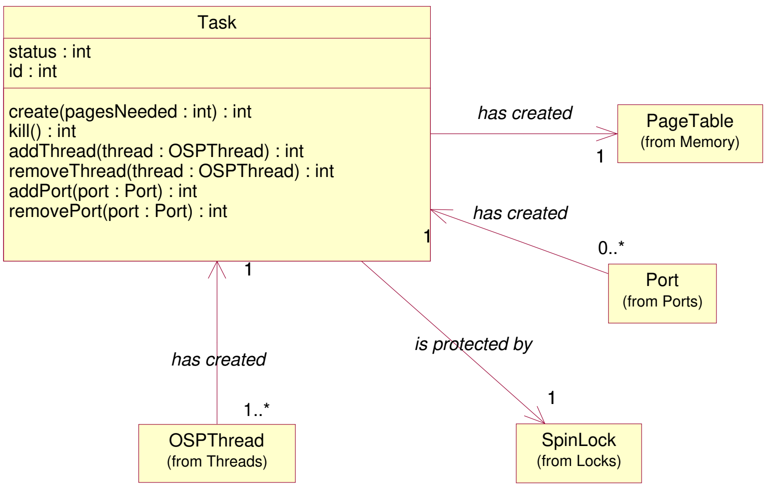

Figure 4.6: *Task class definition*

```
create():
        create thread
        create swap file
        create page table

kill():
        set status field to 'killed'
```

```
kill ports
kill threads
delete swap file
free all used frames
```

When a task is killed, all threads will be killed. It is the responsibility of the thread.kill() method to interrupt all CPUs which are running one of these threads.

4.4.4 Threads

The main difference in the threads module is adding of the lock attributes to class OSPThread[2]. Part of the functionality of these attributes is already explained in section about the locking mechanisms (section 4.4.2). Here it is explained in more detail.

Synchronization Each thread is protected with by spin lock. This lock will be acquired a lot, since threads are accessed from the whole system, as can be seen in the operating system collaboration diagram (see appendix B.4). Threads are suspended and resumed in almost every module and almost every system call requires access to one or more thread objects.
The ready queue and the processor queue also need to be protected against multiple access, so two static spin locks are used. The ready queue contains a list of threads waiting to be scheduled and the processor queue is a list of all processors and the threads running on each of them. The student has to make sure no deadlock can occur while holding one of these two locks.

Scheduling As in the current OSP, the student will be free to implement any kind of scheduling algorithm. So there will be done no error checking on the order in which threads are dispatched. It is important though that the student uses interprocessor interrupts to remove a thread from another processor. This should be checked by the IFL module.
The static method schedule() implements the scheduling algorithm and is called by the timer interrupt handler (a new class TimerInterruptHandler is added). The static method dispatch() is called when a thread should be run, as already determined in the schedule() method. After the timer interrupt handler has called schedule(), it is possible that an interprocessor interrupt is sent to one or more processors. The other processors will call the interprocessor interrupt handler (class IPIInterrupthandler), which calls dispatch(). By using a static variable, for instance the ready queue, the processor can determine which thread should be executed (for example, the first thread in the readyQueue). So dispatch() only removes the current thread from the processor and starts a new one.

Killing a thread It is possible that a thread running on a CPU is killed. In that case, after killing the thread, an interprocessor interrupt (IPI) should be sent to the processor running the killed thread. This will cause the IFL module to put a new event on the event queue, which simulates the IPI (as for all interrupts). The student can implement gang scheduling using IPIs.

[2]The name ThreadCB is changed into OSPThread, TaskCB in Task and PortCB in Port

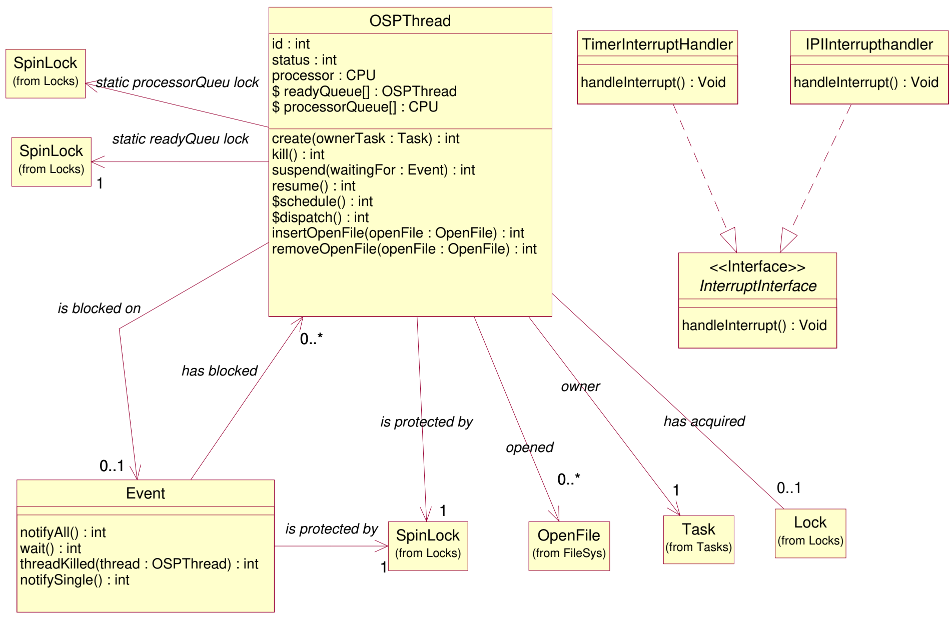

Figure 4.7: *OSPThread class definition*

It is important to keep in mind that a thread can always be killed by another processor. Suppose a page fault occurred on one processor and the page fault handler is called. Maybe, a thread on another processor wants to kill the thread handling the page fault. The thread will be killed, but it is still busy handling the interrupt. It will keep running until it suspends itself (interrupts do not happen while a processor is in kernel mode). Thus it will start swapping in a the page even though the thread was already killed. Thus an I/O event will be added to the thread, specifying the thread is waiting for the page to be swapped in.
To avoid this situation, the status field of the thread is used. When a thread is killed, the status field will be set to killed. Before the page fault handler starts swapping in the page, it should first check to see if the thread was killed or not.

Attributes Here is a list of some attributes in class OSPThread and their use.

status	One of (running, suspended, waiting or killed).
inKernel	True if the thread is running in the kernel.
processor	The processor which is running the thread.
readyQueue	Static list of threads waiting for an processor to run on.
processorQueue	Static list that keeps track of which thread is running on each processor.
acquired lock	List of locks acquired by this thread.
opened	List of files opened by this thread.

Class Event is explained in section 4.4.1.

Methods To complete this section about the threads module, here are the responsibilities for the methods, described in pseudo code (most of the times, the acquiring and releasing of locks are left out).

```
create():
        add thread to task
        add thread to ready queue

kill():
        acquire spin lock
        set status to killing
        release spin lock
        close open files
        remove thread from task
        remove thread from event
```

When a thread is created, using the create() system call, the new thread will have the same owner task as the thread which issued the system call.

```
suspend(event):
        set event attribute in the thread
        call dispatch()

resume:
        remove thread from event
        add thread to ready queue
```

The only thread allowed to suspend a thread, is the thread itself. A thread can be killed though by another thread. Method resume() should only be called when the thread was suspended before.

```
dispatch():
        remove current thread from processor
        if the thread not was killed or suspended
                add the thread to the ready queue
        remove the next thread from the ready queue
        run this thread

schedule():
        execute scheduling algorithm
        send IPIs
        call dispatch()
```

4.4.5 Ports

Ports are much the same as in the current OSP. A semaphore is used to protect the port, mainly consisting of a buffer containing messages. Sending and receiving messages to and from a port looks very much like the producer/consumer problem. This problem describes a situation in which one or more producers fill a buffer with certain products, one at a time. One or more consumers remove these products from the buffer, one at a time. One can find some solutions to this problem using 3 semaphores in [4] and [9]).

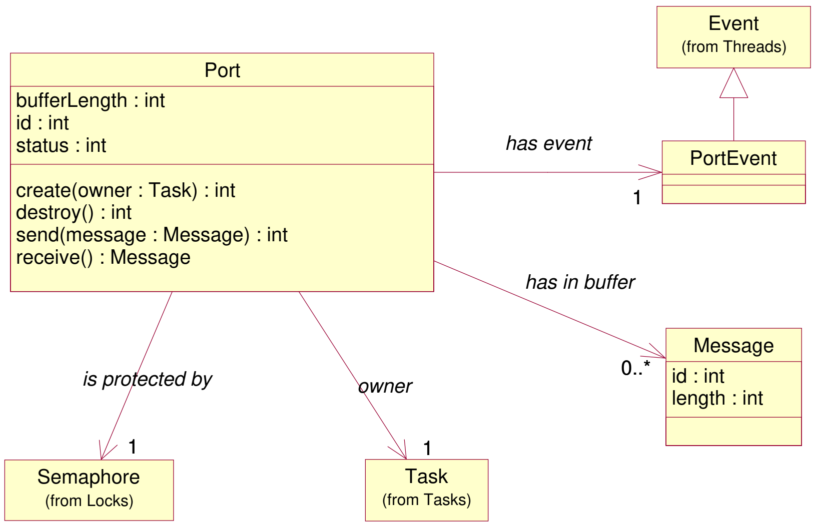

Figure 4.8: *Port class definition*

There is a small difference though which does not allow these solutions to work in the case of the ports buffer, where message are produced and consumed. A message is not of a fixed length, thus the number of messages fitting in the buffer is not fixed. So the amount of free space in the buffer depends not only on the number of messages in the buffer, but also on the length of each of these messages. Therefore these solutions are not used. It would have been interesting though to implement one of these algorithms. Here a simple solution is chosen in which a single semaphore is used to protect the buffer of messages inside the port.
There not much more to explain about this module. So, only the responsibilities for each of the methods are given (again, in pseudo code).

```
create():
        add port to task
kill():
        call PortEvent.notifyAll()
        remove port from task

send(message):
        if the message fits in the buffer
                add message to buffer
                call PortEvent.notifySingle()
        else
                suspend thread on the port event

receive():
        if there is a message in the buffer
                remove message from buffer
```

```
        call PortEvent.notifySingle()

else
        suspend thread on the port event
```

4.4.6 Memory

In OSP, the main difference between a uni- and multiprocessor system in the memory module is the locking mechanism which protects the frame and page tables. Also the page replacement algorithm needs some changes, but in OSP, the student is free to implement any kind of algorithm, thus no attention is paid to this part of the memory module.
The main functionality the memory module implements is providing the system with virtual memory. Swapping is the responsibility of the page fault handler. A second responsibility this module has, both in a uni- and a multiprocessor system, is to protect a page when a disk is copying blocks to or from memory. An extra problem in a multiprocessor system is of course that not only the page in memory needs to be protected against parallel reading and writing, but the other data structures are vulnerable to this kind of access as well.
As for simplifications in the memory module concerned, in both OSP as SMP OSP, no memory sharing between tasks is possible. Only threads owned by the page's owner may read and write from and to the page. The only exception is when a page needs to be swapped out. In this situation a thread from another task can swap out a page in order to use the frame.
Next, the way in which synchronization is realized in the memory module will be discussed. Afterwards especially the classes PageTableEntry and FrameTableEntry will be described in more detail. At the end of this section the process of swapping a page is described in more detail.

Synchronization When processors are referring to addresses via the MMU, and the MMU needs to set the dirty or referenced field, no locks are needed. Assumed is that the hardware will provide atomic memory read and write operations. In SMP OSP, this is simulated, by making sure the setDirty() and setReferenced() methods can be executed as a single atomic operation, using some Java synchronization method.
No synchronization between the MMU and the operating system is needed, because assumed is that the MMU can read the valid field and set the referenced and dirty field in a single atomic operation. When the operating system wants to swap a page out, it can do so by first setting the valid field from the page to false. The MMU will then no longer try to set the fields referenced or dirty from the page object.
Though a page will be protected against multiple threads wanting to read or write a file, it is not protected against multiple MMUs wanting to read or write a word. When a page is valid, thus in main memory, a MMU can always access the page, even though a disk is accessing the page as well. It's up to the software running on top of the operating system, to synchronize this kind of access.
On the other hand, synchronization is needed between threads which need to write or read a file, or swap a page. Therefore, two locks are associated with each page table entry: a spin lock and a read/write lock. The spin lock is to protect the PageTableEntry data structure, so attributes like valid and frame

cannot be accessed by two processors at the same time. The spin lock will not be hold on to for long because it only needs to be acquired when one of the attributes is accessed. The read/write lock though, allows several disks reading from the page in parallel, but allows only one disk to write data to the page in memory. Since I/O operations can take some time, the read/write can be hold on to for a longer period of time. How this lock should be used will be explained later in this section.
The frame table is protected with a single spin lock. The reason is that when a frame needs to be found for swapping in an invalid page, maybe all frame table entries are checked to see if there is one free. Searching for a free frame table entry would become much more complex if each frame table entries has its own lock.

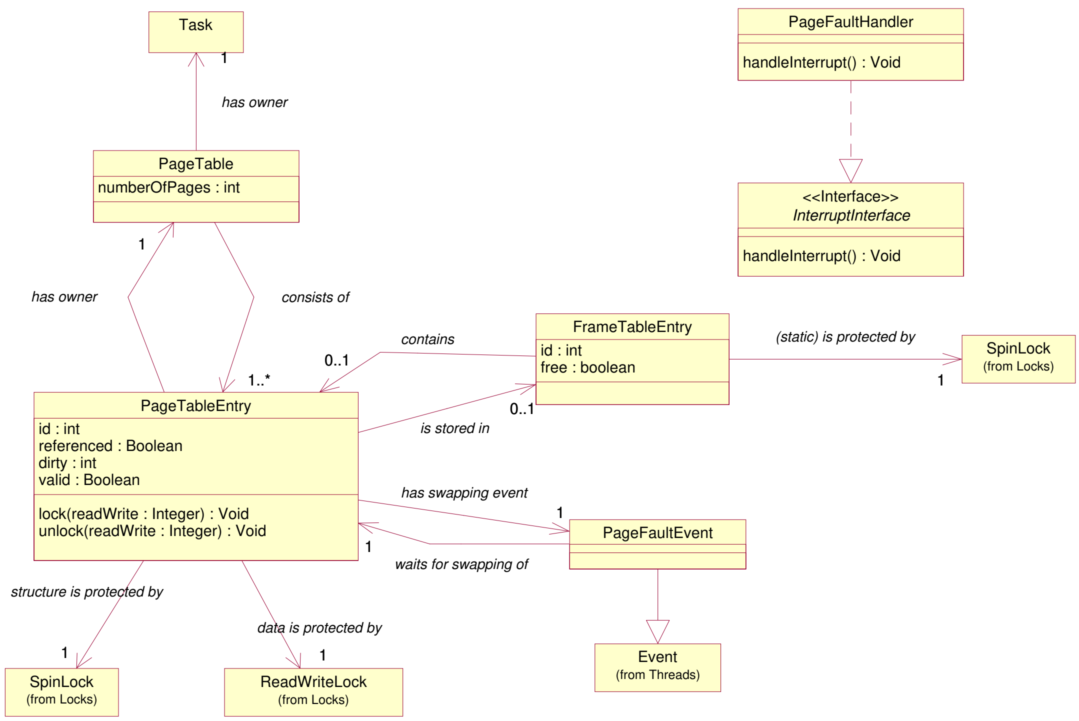

Figure 4.9: *Class diagram of the memory module*

Page & frame table It is important to make the distinction between a page table and a frame table clear to the student. A page table is a data structure each task creates when the task is initialized. For each page used by the task a page table entry is defined. This page table entry gives information about were the page is located, in main memory or not, and whether it was referenced or not. The page table is used by the MMU in order to calculate the physical address from a certain virtual address.
A frame table is a data structure in the operating system which gives information

about each frame in main memory. It is used only by the operating system, not by the MMU. A frame table entry holds a reference to the page stored in the associated frame in main memory and specifies whether or not the frame is reserved by a task to swap in a page.
A page can be in several states. To recognize in which state a page is, the following attributes are defined:

PageFaultEvent	If this is not null, the page is being swapped in.
valid	If this is true, the page is in main memory.
read/write lock	This lock should be acquired before moving data from or to the page. If the page needs to be swapped out, the lock should be acquired as well, using acquire().

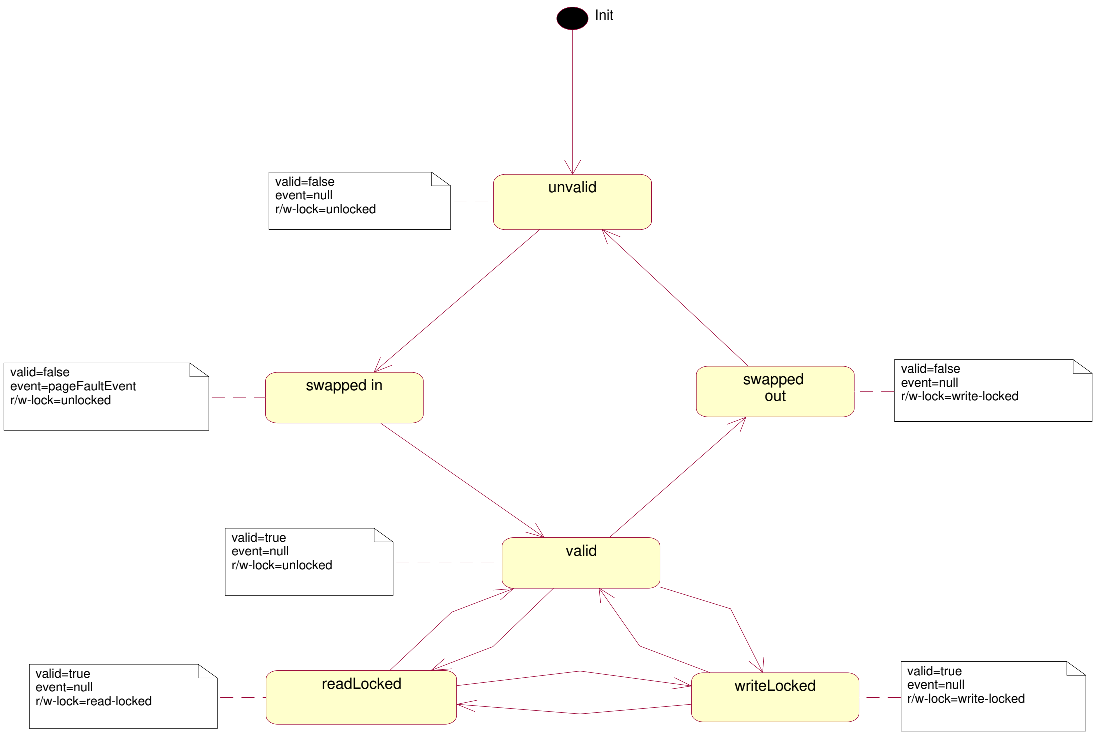

Figure 4.10: *State diagram of a page table entry.*

Figure 4.10 shows which values these attributes have in each of the states. By reading the values of the attributes valid and event, the state of the page can be determined.
The differences with the current OSP are not that big. In the current OSP an attribute lockCount is used to specify whether or not the page is locked. Some other attributes have been removed from FrameTableEntry because the information stored in these attributes is also stored in PageTableEntry.

Swapping Next, the swapping of a page will be examined more closely. This will give insights in how the read/write lock is used, what kind of methods need to be defined and what their responsibility is. Since reading or writing from or to a file is also part of swapping, this paragraph covers all functionality of the memory module.
Below the swapping of a page is expressed in pseudo code. This code will be explained more closely afterwards. The acquiring and releasing of the spin lock is left out, this would take away the overall view. The numbers are used to reference to the pseudo code from the text in which swapping is explained step by step.

```
MMU
     (1) refer(read/write, memoryAddress)
          atomic (
               if page is valid
                      set referenced/dirty field;
          )
          if page is not valid
               start interrupt;

PageFaultHandler:
     (2) handleInterrupt()
         (3)   if page.event not is null
                      suspend thread on event
               else
                  (4) set new event to page.event
                  (5) search for a frame to store the page
                      reserve this frame
                  (6) lock the to be swapped out page (pageOut) in the frame
                      for writing, using the read/write lock
                  (7) set pageout.valid to false
                      if pageout.dirty is true
                           swap this page out
                      unlock pageOut
                  (8) acquire write lock for page
                      swap page in
                  (9) release write lock for page
                      resume all threads blocked on event
```

(1) First, the MMU refers to a certain address. It checks the page table to see if the referenced page is stored in main memory or not by reading the valid field. If the page is stored in main memory and the reference concerns a write action, the dirty bit is set. All these actions need to be executed as an atomic operation. A small example shows what can go wrong when the operation is not atomic: the MMU reads the valid bit, which is true. Then the interrupt handler running on another processor sets the valid bit to false and reads the dirty bit, which is false. Thus it seems the page does not have to be written to the swap device. Then the MMU sets the dirty bit, and the page will become corrupted.
(2) Suppose the page (called pageRef) is not valid. In that case the interrupt handler needs to be called. The general interrupt handler will call the page

fault handler. This method is responsible for swapping in the page holding the data from the referenced memory address. The page will be swapped in from the swap device. Each task has a swap file in which all pages are stored. This swap file is stored on the swap device.
(3) First, the page fault handler checks the state of pageRef by reading the event attribute. As can be seen from the state diagram 4.10, if the event is not empty, then the page is already being swapped in. In that case the page fault handler has to wait for this event to happen by calling its wait() method.
(4) Suppose there is no event specified. Then the page is either invalid or being swapped out. Thus, the page needs to be swapped in again. A new page fault event is created. Before moving the page from the swap device to main memory a frame needs to be found to store the page in.
(5) The page replacement algorithm is responsible for finding such a frame, it is part of the page fault handler. The student can implement this algorithm as he or she wishes. Attributes referenced and dirty defined in each page can help to decide which page to swap out.
(6) When a frame is found, the page which is currently located in this frame needs to be swapped out. This means the page has to be written to the swap file. First the page is locked (write lock). This way non of the other threads can access this page. Once the page has been locked, other threads trying to lock the page will get suspended.
(7) Next the page (pageOut) can be set to invalid and I/O can be started by calling write() from the file system. This method is responsible for writing the page to the swap file and blocks the currently running thread. After the threads becomes awake again, the page (pageOut) can be unlocked.
(8) Then pageRef can finally be read from the swap file. First the write lock needs to be acquired though. This is probably available, because the only thread which could have had this lock was the thread swapping the page (pageRef) out of memory. The current thread gets suspended again after calling read() from the file system.
(9) After the thread wakes up, the page is stored in main memory. The read/write lock can be released, the threads blocked on the page fault event can be resumed and valid can be set to true.

Lock and unlock methods Method lock() defined in PageTableEntry is called from the device module when a device wants to move a block of data between a page and a file. Lock() has acquire the read/write lock, but first it must make sure the page is valid. If the page is not valid, it has to be swapped in. This is done by calling the page fault handler. If the page is valid, then only the read/write lock needs to be acquired. Method unlock() only has to release the read/write lock again.

4.4.7 File system

The file system will not be discussed in much detail. As stated in section 2.4.6, the links between the file system and the rest of the system are very limited. The main functionality the file system provides to the rest of the operating system is used for swapping.
To protect the file system only one lock is used. This is a mutex lock because updating the file system structures can take some time. Each method within

the file system should first acquire this lock and release it again once it has completed its actions. Thus only one thread at a time is allowed to execute in the file system. Besides this mutex lock, each open file object has a spin lock. This way an open file can also be accessed directly by the device module, without first accessing the file system. Inside the spin lock it is not allowed to acquire another lock (see 4.4.2).
Besides the addition of these locks, some responsibilities have been removed from the file system. In SMP OSP an attempt is made to keep similar method calls in the same module. This results in a more modular design, thus easier to understand and easier to change.
The file system is no longer responsible for suspending a thread when it starts an I/O operation. On the other hand, it is should remove the open file from the thread after the I/O has taken place.

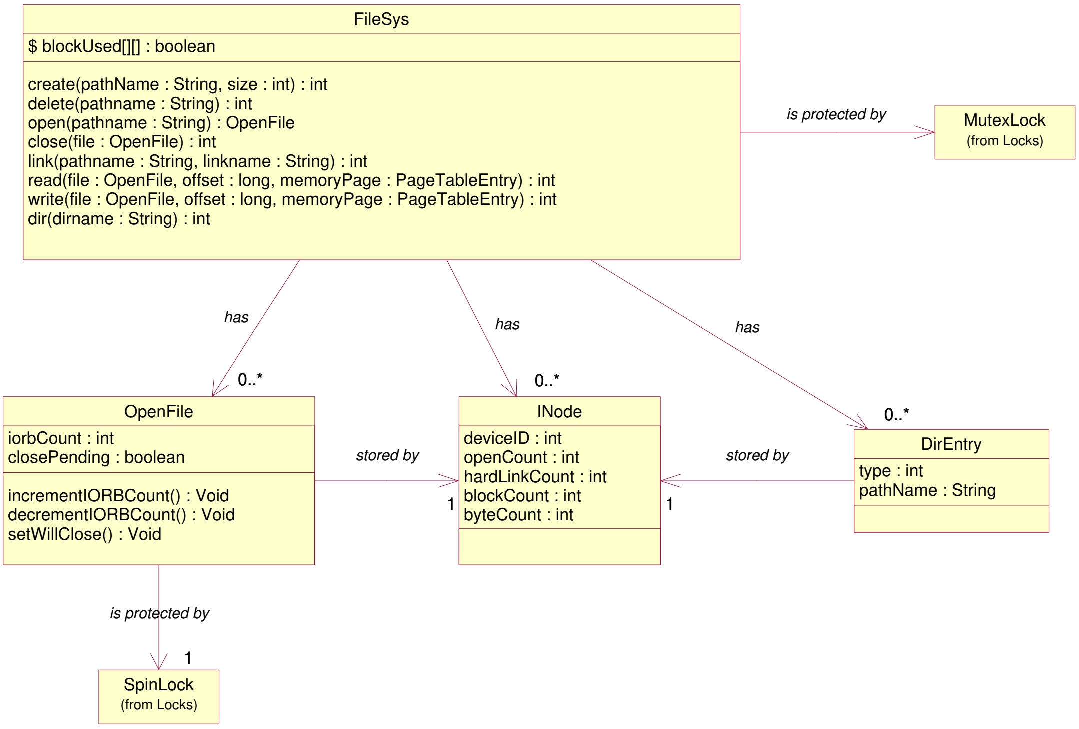

Figure 4.11: *File system class definition*

The most important methods are read() and write(). Read() reads a block from a file, stored on one of the disks and copies it into a specified page in main memory. Write() copies data from the specified page to a block on one of the disks.
The opening and closing of open files stays more or less the same. The open file is not removed by the disk interrupt handler anymore. This is up to the user (thus the simulated software, thus the event engine) or the page fault handler when swapping. This results in the next pseudo code. (the acquiring

and releasing of the locks is left out)

```
FileSys:
        open(file):
                update internal data structures
                add the open file to the current thread
        close(file):
                remove the file from the thread
                if IORB count from the open file > 0
                        set willClose in the open file
                else
                        update internal data structures

OpenFile:
        incrementIORBcount():
                increment the IORB count from this open file

        decrementIORBcount():
                lower the IORB count
                if count == 0 and willClose is true
                        close this open file
```

The difference between reading and writing is small, because no actual data is moved from a device to the main memory or the other way around. Therefore, no distinction will be made in the description between these two methods.

```
write/read(file):
        FileSys:
                update internal data structures
                create an IORB
                enqueue IORB to device
```

The only thing the file system has to do is start the I/O is call the device module. This takes care of suspending the thread and incrementing the IORB count. More information about the device module and enqueueIORB() can be found in the next section. The other system calls from the file system will not be discussed.

4.4.8 Device

As can be read in the section about the file system, some responsibilities have been shifted from the file system to the device module. Also, some have been shifted from the disk interrupt handler to the device module. This results in a more modular design. Which responsibilities have been shifted is explained in the description of each of the methods.
Another important difference with the current OSP is that Disk has been moved from the operating system to the hardware module. The reason has been explained before. Since Disk is no longer a subclass of Device, each device holds a reference to a disk. To find out which device is associated with each disk from hardware an extra method is needed which is called getDevice(). This is a static method.

A last change in the device module is that IORB is not a subclass anymore of Event, It holds a reference to an event, which results in a more transparent design. When a disk has finished a certain I/O operation, the associated device removes the IORB from its queue and resumes all thread waiting for the event.

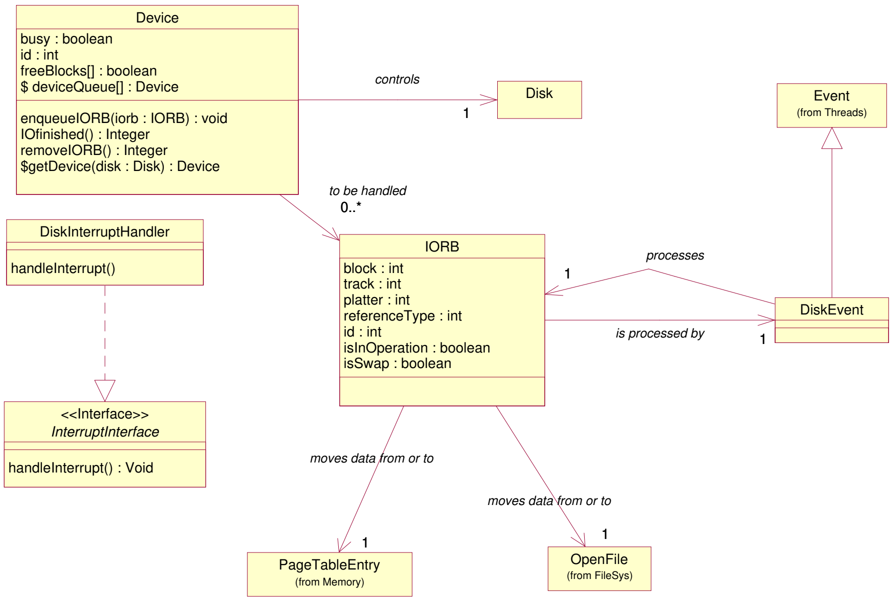

Figure 4.12: *Device class definition*

Next, each method will be explained more closely describing all responsibilities in pseudo code.

```
getDevice(disk) (static)
        return the device associated with this disk

enqueueIORB(IORB):
        increment IORB count in the open file
        lock the page, specified in the IORB, by calling page.lock()
        add IORB to queue
        if disk not is busy
                start disk IO
        call DiskEvent.wait()
```

Method IOfinished() is called by the disk interrupt handler and should do the opposite of what enqueueIORB() does:

```
DiskInterruptHandler:
        handleInterrupt():
                check which disk started interrupt
                call IOfinished() of associated device
```

```
Device:
        IOfinished():
                remove IORB from queue
                decrement IORB count from open file
                resume threads waiting for it
                if there is another IORB
                        start disk IO
```

The responsibility of suspending the thread has been moved from the file system to the device module. The disk interrupt handler almost loses all its responsibilities, these have been moved to IOfinished() in class Device. There is one method left:

```
removeIORB(IORB)
        if IO not already started
                remove IORB from queue
                decrement IORB count from open file
```

The method purgeIORB has been removed. In the current OSP this method was called when a thread was killed while waiting on an IORB. This responsibility has been moved to the class DiskEvent, like for all subclasses of the event class. When a thread is killed, it calls the threadKilled() method from DiskEvent. This method makes sure the thread is removed from the waiting thread list. If it was the only thread waiting, removeIORB() in Device is called to cancel the I/O operation.

4.5 Error checking

The use of Java threads running concurrently make it more difficult to do error checking in the IFL module. The following problem arises (see figure 4.13 for a visualization of the problem). Suppose two OSP threads are running on separate CPUs. Thread 1 wants to send a message to a port and thread 2 wants to receive a message from the same port. The ports buffer is empty and the threads will run concurrently. Thread 1 calls send() defined in the IFL module. The IFL module then calls do_send() defined in the students module. This method adds the message to the empty buffer. Thread 1 does not return to the IFL module yet, because at this moment thread 2 will start executing. Thread 2 will also call the IFL module, which will call the students module. The message just added to the buffer will be removed and thread 1 will return to the IFL module. Now the IFL module does error checking. There is no message in the IFL copy of the ports buffer, because thread 1 has not returned yet. But the students method did return a message. This seems incorrect so the IFL module will generate an error, while actually the students code is correct.
This example shows that the way of doing error checking will need to be changed. There are several solutions possible to this problem. A first solution would be to remove some error checking. For example in the ports module, the checking of the order in which messages are received could be removed. This is not the most wanted solution, because one of the main characteristics of OSP is the error checking. The responsibility of checking the students code would be shifted more to the person who grades the assignments.

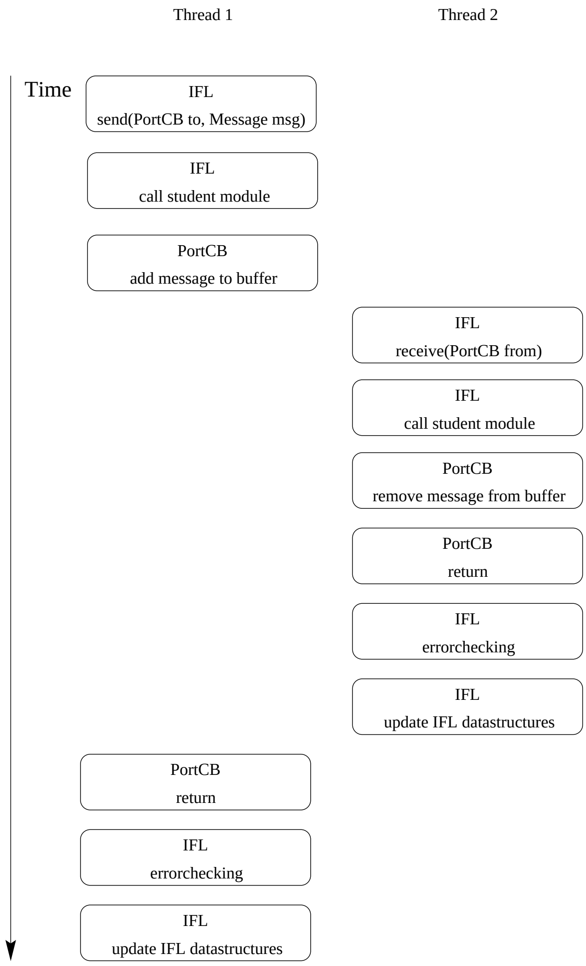

Figure 4.13: *The order in which messages are sent and received cannot be checked by the IFL module using the current error checking mechanism.*

A second solution would be to move data structures from the student module to the IFL module. In the ports module, the student would have to call certain methods provided by the IFL module to add and remove messages. Each time the attribute is updated, the IFL can notice this. This way the IFL attributes stays up to date. In combination with the next option the IFL can check which message was removed or added by each thread.

The third option is to store a list of the methods called from the students module. This list can be attached to each OSP thread in which all called methods and parameters are stored. Each time a method is called in or via the IFL module, the attached object is updated. When the student method returns, the IFL can check the attached object to find out what methods were called and what parameters were used. This is the way it is done in the memory module in the current OSP system.

For each module and system call a choice has to be made about what error checking method to use and which attributes should be moved or copied into the IFL module. Every system call requires a different way of doing error checking, no uniform way of dealing with error checking can be given.

5 Conclusions

In the previous chapter a design was outlined for SMP OSP, the multiprocessor variant of OSP. At the beginning some characteristics were defined which the system should have. The first one is that SMP OSP should be simple but realistic. By focussing on the operating system and not on the IFL module, a realistic system was designed. The IFL module was designed only after the operating system design was finished. This resulted in a realistic design limited by the simplifications chosen at the beginning of the design. Without these simplifications the system would become too complex and the goal of the OSP tool would not be reached.

By choosing for concurrently running Java threads, another step was made towards a realistic simulation. The student will notice that several processors (threads) are running concurrently so he or she must take extra care in synchronizing the data structures inside the operating system. There is also a disadvantage to this choice. The use of Java threads running concurrently do not make it easier for the OSP programmer. Sometimes it is hard to find out how these threads will behave, because several threads are running at the same time.

The second characteristic defined was that the system should be modular. One important improvement is that the three main modules, hardware, operating system and event engine, have been separated. In the current system the distinction between these modules was not always clear but in SMP OSP these three modules were designed separately. This also becomes clear from the process of designing SMP OSP. First the hardware module was made, only then the operating system.

Though these changes are not that big, they result is a much clearer system, not only to the OSP programmer, but especially for the student, for whom this program is made. Operating system are a topic with some difficult aspects, which is the reason that tools like OSP have been developed. Therefore it is important to give the student a system with a clear design and a strict separation of modules. Not only is it easier for the student to learn how to work with the program, also the system will look more realistic.

In the SMP OSP design the responsibilities of each of the modules is defined and it is made sure that similar responsibilities, like locking and unlocking, are put in the same module, or even the same method. This results in clear distinct assignments, which is very useful from the point of view of the teacher. The teacher can much easier skip an assignment and adapt the course to his or her wishes.

The third goal was to create a transparent design. Though the student will not know how the event engine and the IFL module are implemented he or she will have a better overall view of the system. The interface with the IFL module and the event engine is kept very small, so the student will not start to wonder

how these modules work, at least less often. Especially the changes in the use of events have helped to reach this goal.
This design is not completely new, but rather a redesign of the current system. A lot of code from the current OSP program can be used to implement SMP OSP. This was also one of the goals defined at the beginning: to build a design which uses as much code from the current program as possible. How much code actually can be copied can only be determined after implementing the system. But it seems that a programmer who knows how the current system is build, should be able to implement SMP OSP in a couple of months.
Of course SMP OSP is not a complete copy of the current OSP. It has turned out that the biggest change was the addition of the locking module and the synchronization of all the objects. Thinking about programs that run several threads in parallel is not easy. Sometimes it seems that solving a problem in one place always results in the appearance of another problem in some other place. Therefore a general way of handling synchronization and deadlock avoidance was given. It is up to the programmer and the student to follow these rules and solve more specific problems during the implementation.
The extra module forms an interesting extra assignment for the student. Synchronization has always been a difficult topic for students, but he or she can get better insight in the problems involving synchronization implementing this extra module and of course using the locking mechanisms in the rest of the system. This is one of things what makes SMP OSP interesting, the student will see how synchronization methods are used in a system and also has to use them him or herself. This is also what makes SMP OSP more complex.

To do

This is only the design of SMP OSP, hopefully an implementation soon will follow. In this design 7 operating system modules are given, but some new ones can be added in future. A module which provides access to a network would be very interesting. In the first version of OSP, the one written in C, such a module was part of the system. It also contained a module for managing resources, complete with a deadlock avoidance or deadlock detection. Such a module can also be implemented in the current OSP and SMP OSP. Disks and ports can be the resources which can be locked by user programs (simulated by the event engine). Some change in the system are necessary though, because then user programs must be able to lock these resources, which is not the case in the current system and SMP OSP. Now, only within the operating system data structures can be locked. New system calls have to be implemented which lock the resources.

The implementation of this SMP OSP design will be a good tool for students to get acquainted with multiprocessor operating system aspects. It is a simplification of an operating system, even though synchronization within the operating system have made the system more complex. But it is more realistic, it is well structured and thus easier to understand. The student can concentrate on the things the tool was meant to be for: work on a more or less real operating system, and getting acquainted with the main aspects of an operating system.

Bibliography

[1] Scott Smolka, Michael Kifer *OSP, An Operating System Project* Addison-Wesley, 1991

[2] Wayne A. Christopher, Steven J. Procter, Thomas E. Anderson *The Nachos Instructional Operating System* Available from http://http.cs.berkeley.edu/ tea/nachos/nachos.ps. (Link checked August 13, 2000)

[3] Andrew S. Tanenbaum, Albert S.Woodhull *Operating Systems, Design and Implementation* Prentice Hall Inc, New Jersey, 1999 Second Edition

[4] William Stallings *Operating Systems, Internals and Design Principles* Prentice Hall Inc, New Jersey, 1998 Third Edition

[5] James Rumbaugh, Ivar Jacobson, Grady Booch *The Unified Modeling Language Reference Manual* Addison Wesley, 1999

[6] Doug Lea *Concurrent Programming in Java, Second Edition, Design Principles and Patterns* Addison Wesley, 2000

[7] Ben Catanzaro *Multiprocessor System Architectures* Sun Microsystems, Inc, 1994

[8] Mukesh Singhal, Niranjan G. Shivaratri *Advanced Concepts in Operating Systems, Distributed, Database and Multiprocessor Operating Systems*, McGraw-Hill Inc, 1994

[9] Andrew S. Tanenbaum *Structured Computer Organization, Fourth Edition* Prentice Hall, New Jersey, 1998

[10] Harold S. Stone *High-Performance Computer Architecture* Addison Wesley 1990 Second Edition

[11] M. Flynn *Computer Organizations and Their Effectiveness* IEEE Transactions on Computers, September 1972.

[12] Uresh Vahalia *Unix Internals, the New Frontiers* Prentice Hall, New Jersey, 1996

[13] Richard Y. Kain *Advanced Computer Architecture, A Systems Design Approach* Prentice Hall, New Jersey, 1996

[14] *Intel Multiprocessor Specification* Intel Corporation, 1997

[15] Alan Cox *An Implementation Of Multiprocessor Linux* Available from http://www.uk.linux.org/SMP/title.html, 1995 (Link checked August 13, 2000)

Appendix

A Parameter file

```
[General]
SimulationOutputFlag 1
UserOutputFlag 1
NumberOfSnapshots 40
LogFile OSP.log
SimulationLength 60000
AverageTicksBetweenEvents 10
FileEventPercentage 30
MemoryEventPercentage 30
PortsEventPercentage 40

[Tasks]
AverageNumberOfTasks 4
TaskLifeExpectancy 500
ServerTaskPercentage 20
MaxTasks 6

[Threads]
ThreadLifeExpectancy 500
AverageNumberOfThreads 3
MaxThreadsPerTask 5

[FileSys]
FilesEventPercentage 60
DirsEventPercentage 40
CreateFilePercentage 20
OpenFilePercentage 20
CloseFilePercentage 5
DeleteFilePercentage 5
ReadFilePercentage 20
WriteFilePercentage 20
LinkFilePercentage 10
CreateDirPercentage 30
DeleteDirPercentage 10
ListDirPercentage 60

[Memory]
ReadPercentage 50
```

```
WritePercentage 50
PageAddressBits 6
FramesOfPhysicalMemory 32

[Devices]
NumberOfDevices 2
Disk1NumberOfPlatters 4
Disk1TracksPerPlatter 10
Disk1SectorsPerTrack 32
Disk1RevolutionsPerTick 5
Disk1SeekTimePerTrack 1

[Ports]
PortCreatePercentage 30
PortDestroyPercentage 10
PortSendPercentage 15
PortReplyPercentage 15
PortReceivePercentage 30
MaxPortsPerTask 5
MeanMessageLength 5
PortBufferLength 7
```

B UML diagrams

B.1 Event engine class diagram (current OSP)

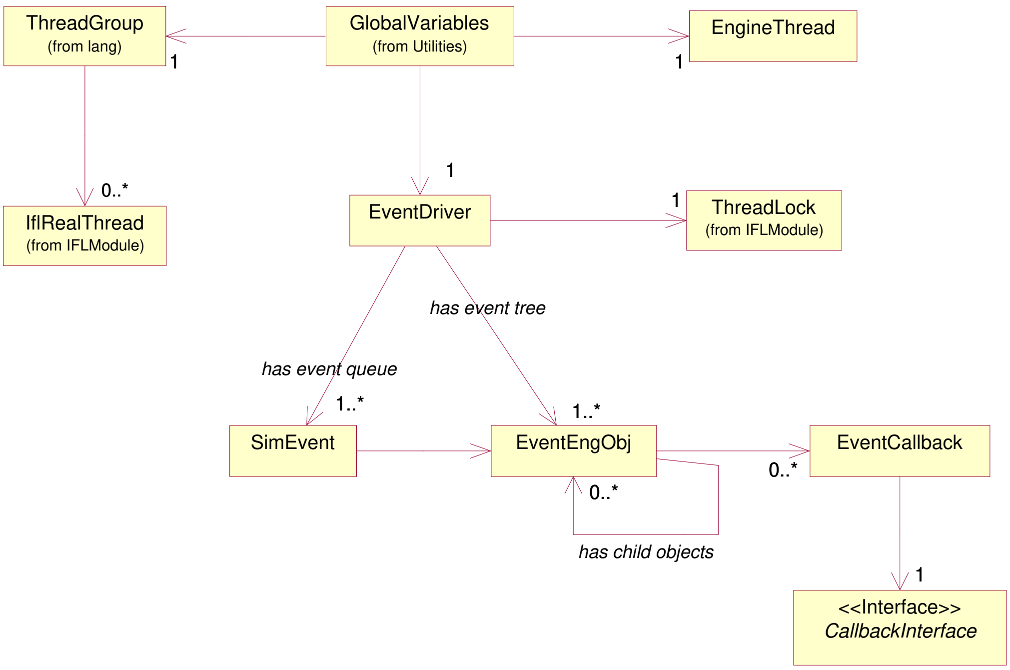

Figure 1: *Class diagram of the event engine as defined in the uniprocessor version of OSP. Attributes and methods are left out.*

B.2 Current operating system class diagram

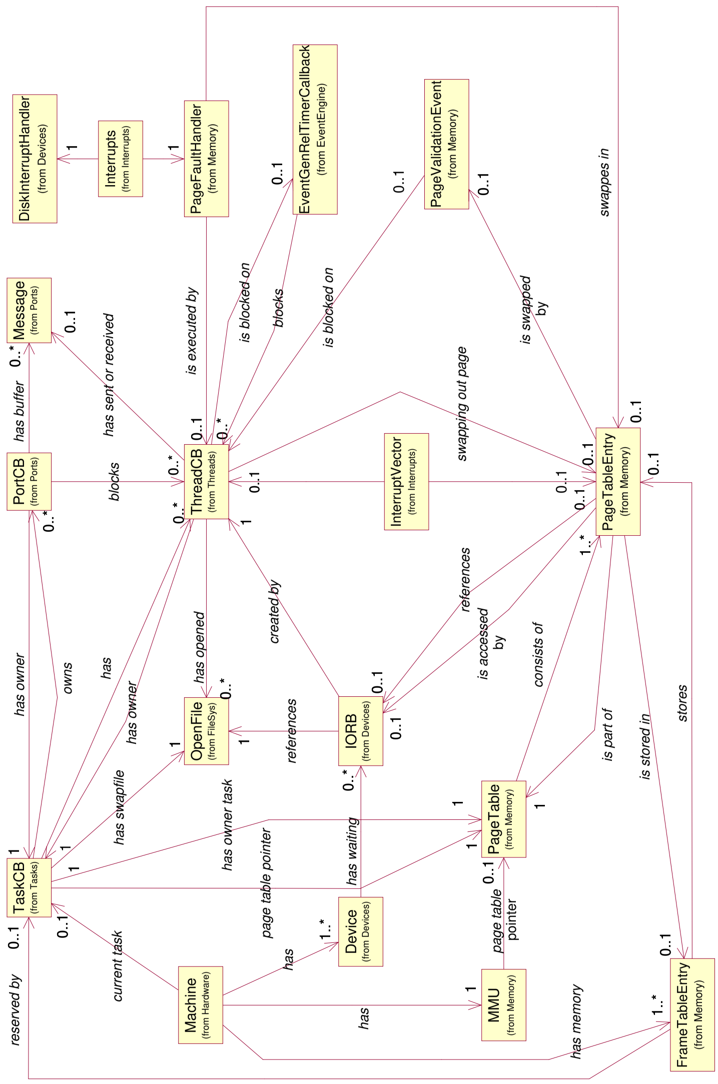

Figure 2: *Class diagram of the operating system. Associated classes from the IFL module and the students module are shown as one class box. Module FileSys is left out.*

B.3 Current operating system collaboration diagram

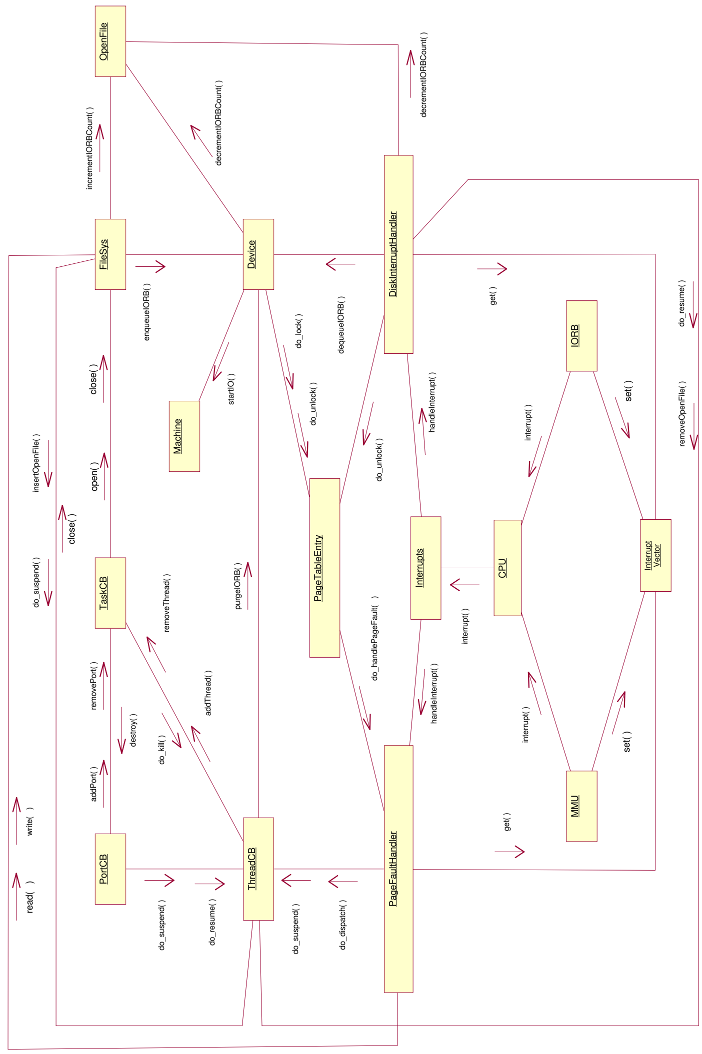

Figure 3: *Object collaborations in the operating system. Associated classes from the IFL module and the students module are shown as one class box.*

B.4 SMP operating system collaboration diagram

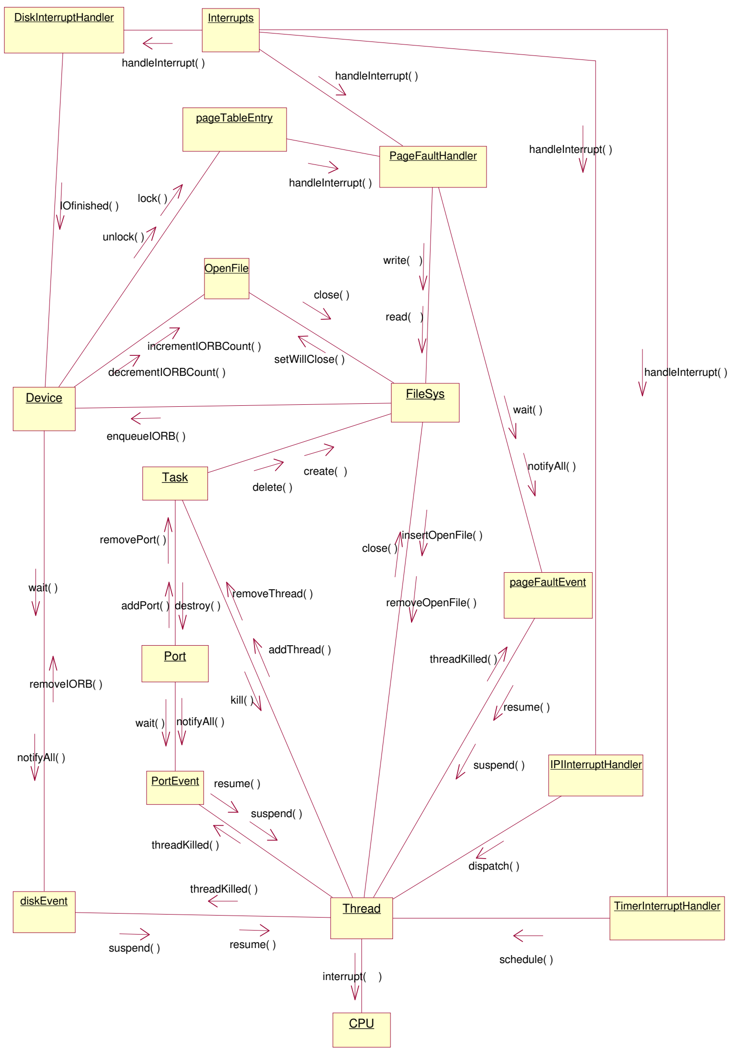

Figure 4: *SMP OSP collaboration diagram of the operating system module (without IFL).*

Acronyms

OS	Operating System
OSP	Operating System Project
SMP	Symmetric Multiprocessor
UML	Unified Modeling Language
TLB	Translation Lookaside Buffer
PTE	Page Table Entry
MIMD	Multiple Instruction Multiple Data
UMA	Uniform Memory Access
NUMA	Non Uniform Memory Access
IORB	I/O Request Block
I/O	Input/Output
CPU	Central Processing Unit
IC	Interrupt Controller
MMU	Memory Management Unit
IPI	Interprocessor Interrupt