ABSTRACT
The paper describes a mini-kernel project in the context of a Concurrent Programming course. The goal of the project is to implement Java monitors and interrupt handling. The platform for the project is an FPGA board developed initially at EPFL for Computer Architecture courses.

1. INTRODUCTION
Concurrent Programming can be taught in a CS bachelor curriculum as part of an OS course, or as a stand-alone course; EPFL has adopted the latter model. Such a Concurrent Programming course, apart from the theoretical part where students learn the basic synchronization mechanisms (locks, semaphores, monitors, etc.), often includes small practical projects where students need to apply the concepts they have learned. Our experience with such projects is that there are always students that produce code of poor quality, e.g., code with busy waiting or code that can deadlock. This clearly shows the problems that some students have in understanding how synchronization primitives should be used. Those students probably look at synchronization primitives in the same way they look at functions provided by some standard libraries in the context of sequential programming, i.e., without an effort trying to understand the specificity of concurrent programming. We believe that exposing concrete implementation of synchronization primitives might help those students to overcome their conceptual difficulties, while helping other students to have a more deep understanding of synchronization.

We describe here such a mini-kernel project for the undergraduate Concurrent Programming course at EPFL. The background of the student is a first year programming course in Java. Therefore, in addition to the classical synchronization mechanisms, the course teaches the mechanisms provided by Java (Java monitors). This leads the mini-kernel project to be naturally “Java oriented”. It allows students to “touch” what Java hides, namely thread scheduling and implementation of synchronization primitives. The project also allows students to be exposed to interrupt handling, a topic that cannot be covered in Java.

A (mini-)kernel project requires an adequate platform. Our hardware platform consists of an FPGA-based board [1] illustrated in Figure 1. The board is a pedagogical initiative of the School of Computer and Communication Sciences of EPFL, developed initially for the Computer Architecture courses. It consists of an Altera FPGA and several IO devices: 2 rows of 8 switches (Fig.1, top right), 4 buttons (aligned vertically, see Fig.1, extreme right), and a matrix of 8x12 LEDs (Fig.1, to the left of the buttons). The input devices (switches and buttons) can be read in busy wait mode or in interrupt mode. The board also includes a clock that can be configured to generate interrupts at a given frequency. The kernel project uses the C language. Compilation is done on a PC, and the code is downloaded on the board using a USB port. Note that some of the students have no C experience before starting this project. We provide these students material to go from Java to C.

Related work: In [6] the authors describe a programming project where the students have to write a user-level thread library, similar to the POSIX pthread library, for a Unix or Linux system. In contrast, our project is stand-alone and Java oriented. Moreover, the use of the primitives described in Section 2 provide to the students high level abstractions that simplify the writing of the (mini-)kernel, allowing a
higher level view on thread implementation. Thread implementation is also reported in [3], but the project does not include preemptive scheduling and implementation of synchronization primitives. Other projects, in the context of an Operating Systems course, cover components beyond thread library, e.g., the Embedded XINU project [4], with the implementation of a tiny file system. Such projects are beyond the scope of a stand-alone Concurrent Programming course.

Paper structure: The paper is structured as follows. Sections 2 and 3 provide the conceptual background that helps the students before starting the kernel project. Section 4 describes the kernel project. A discussion section concludes the paper. All material provided to the students is available online, see [2].

2. AVAILABLE PRIMITIVES

We provide the students with three primitives for writing a kernel. These primitives are almost identical to the primitives provided by the Modula-2 language [8].\(^2\) We describe first the primitive for process creation and the primitive to allocate the CPU to a process. The primitive related to interrupt handling is described later.

2.1 Creating and executing a process

A process is created using the newProcess function:

\[
\text{Process newProcess(void \(*f\), int \(*stack\), int stackSize)}
\]

The first parameter is a function that defines the code of the created process. The second parameter gives the address of the memory space that has been allocated for the process stack. The last parameter specifies the stack size.\(^3\) The function returns a result of type Process, which is a process reference.

If we ignore interrupts (discussed later) processes are actually coroutines. If process p is running, it keeps the CPU until it allocates the CPU to another process, using the transfer function:

\[
\text{void transfer(Process p)}
\]

When process \(p_1\) executes transfer\((p_2)\), then \(p_1\) releases the CPU and \(p_2\) gets the CPU. When later some process executes transfer\((p_1)\), then \(p_1\) resumes execution at the statement that immediately follows transfer\((p_2)\). The mechanism can be illustrated on the following example:

```c
void code_p1() {
    while (true) {
        S1;
        transfer(p2);
    }
}
```

```c
void code_p2() {
    while (true) {
        S2;
        transfer(p1);
    }
}
```

```c
void main() {
    const int size_stack1=...;
    const int size_stack2=...;
    int stack1, stack2;
    Process p1, p2;
    stack1 = malloc(size_stack1);
    stack2 = malloc(size_stack2);
    p1=newProcess(code_p1, stack1, size_stack1);
    p2=newProcess(code_p2, stack2, size_stack2);
    transfer(p1);
}
```

In this example, main creates two processes \(p_1\) and \(p_2\), and then allocates the CPU to \(p_1\). Process \(p_1\) executes statement \(S_1\), and then allocates the CPU to \(p_2\). Process \(p_2\) executes statement \(S_2\), and then allocates the CPU to \(p_1\). Process \(p_1\) executes statements \(S_1\), and then allocates again the CPU to \(p_2\), etc. In other words each process, one after the other, executes one iteration of its while loop. Note that main never gets the CPU back.

2.2 Interrupt handling

The transfer primitive provides a purely coroutine like execution schema unable to handle interrupts. A third primitive, called ioTransfer, is needed:

\[
\text{void ioTransfer(Process p, int interrupt)}
\]

The semantics of ioTransfer has two distinct parts: (a) upon execution of ioTransfer, and (b) when the interrupt specified by the second parameter occurs.\(^4\) Consider Figure 2, and ioTransfer\((p_2, i)\) executed by process \(p_1\). With respect to (a), ioTransfer\((p_2, i)\) is equivalent to transfer\((p_2)\). To explain (b), consider that at time \(t\) interrupt \(i\) occurs, while some process \(p_3\) is running (possibly \(p_3=p_2\)). Assume that \(p_3\) is interrupted between statements \(S_1\) and \(S_2\). Then the interrupt is equivalent to transfer\((p_1)\) executed between \(S_1\) and \(S_2\). The use of ioTransfer is illustrated in Section 3.2.

\[\text{Figure 2: Illustration of ioTransfer}\]

\(^2\)Two out of our three primitives have one parameter less than the corresponding Modula-2 primitives.

\(^3\)The stack grows towards small addresses, which explains the need for the stack size.

\(^4\)We simply refer to different interrupt sources by integers.
Note that interrupts need to be disabled within a kernel. For this, we provide the two functions `void maskInterrupts()` and `void allowInterrupts()`.

3. ILLUSTRATION: KERNEL PROVIDING SEMAPHORES
The primitives `newProcess`, `transfer` and `ioTransfer` are low level primitives, that are not supposed to be used by an application. Their use should be limited to the kernel. This is now illustrated in the context of a kernel providing semaphores (see also [7] for an example in Modula-2, namely the implementation of a mailbox).

3.1 Kernel without interrupts
We start with a simplified kernel that does not provide interrupt handling (interrupt handling will be added later). We first define the kernel interface, and then discuss its implementation.

3.1.1 Kernel interface
Our kernel provides the following interface:

```c
void createProcess(void (*f), int stackSize): This function is very similar to `newProcess`, with small differences. The two parameters of `createProcess` match the first and the third parameter of `newProcess`; the second parameter of `newProcess`, namely the stack address, has disappeared. Indeed, it is the role of the kernel to allocate space for the stack. The second difference is that `createProcess` does not return any result: the process reference, returned by `newProcess`, is kept inside the kernel.

void start(): This function starts the execution of the processes that have been created. The kernel will decide which is the first process to run. Note that at least one process must have been created before calling `start()`.

int createSemaphore(int n): Allocates and initializes the data structure for a semaphore. The semaphore counter is initialized to `n`. Semaphores are referenced by integers. The function returns the identity of the newly created semaphore.

void P(int s): Operation `P` (decrement) on semaphore `s`.

void V(int s): Operation `V` (increment) on semaphore `s`.

void yield(): Since the kernel does not implement time-slicing, this function allows a process to release the CPU (and assign it to another process).
```

3.1.2 Kernel implementation
The main data structure of the kernel is a list of processes ready to be executed. Consider the following functions available for handling a queue: `void addLast(List l, Process p)` (which adds process `p` to the tail of `l`), `void addFirst(List l, Process p)` (which adds process `p` to the head of `l`), `Process removeHead(List l)` (which returns the process at the head of `l` and removes it from the list), and `Process head(List l)` (which returns the process at the head of `l`, or `nil` if the list is empty). Moreover, let `readyList` represent the list of ready processes. Then `void createProcess()` simply becomes:

```c
void createProcess(void (*f), int stackSize) {
    Process p; int *stack;
    stack = malloc(stackSize);
    p = newProcess(f, stack, stackSize);
    addLast(readyList, p);
}
```

If the kernel policy consists of allocating the CPU to the process at the head of the ready list, then `start()` becomes:

```c
void start() {
    Process p;
    p = head(readyList);
    if (p==nil) { //ready list empty
        error message;
    } else {
        transfer(p);
    }
}
```

For a semaphore `s`, let `sem[s].n` represent the counter of the semaphore, and `sem[s].wq` represent the waiting queue of the semaphore. Then `P()` and `V()` are implemented as follows:

```c
void P(int s) {
    Process p;
    sem[s].n = sem[s].n - 1;
    if (sem[s].n < 0) {
        p = removeHead(readyList);
        addLast(sem[s].wq, p);
        p = head(readyList);
        transfer(p);
    }
}

void V(int s) {
    Process p;
    sem[s].n = sem[s].n + 1;
    if (sem[s].n <= 0) {
        p = removeHead(sem[s].wq);
        addLast(readyList, p);
    }
}
```

Note that with this implementation, a process executing `V(s)` keeps the CPU. This is perfectly ok since processes all have the same priority. Finally, here is the implementation of `yield()`:

```c
void yield() {
    Process p;
    p = removeHead(readyList);
    addLast(readyList, p);
    p = head(readyList);
    transfer(p);
}
```
3.2 Handling interrupts
We extend now the kernel to include interrupt handling. We first show the extended kernel interface, and then show how to implement the new functions.

3.2.1 Extended interface
We add one single function to the kernel interface:

```c
void waitInterrupt(int interrupt) ;
```

Note that the function looks like `ioTransfer` (Sect. 2.2), however with one fewer parameter. The missing parameter is the process to which to allocate the CPU. Indeed, selecting this process is not the job of the process calling waitInterrupt. With waitInterrupt interrupt handling is straightforward.

3.2.2 Implementation of the extended interface
Using `ioTransfer` and the `readyList` (Sect. 3.1.2), the implementation of `waitInterrupt` is almost straightforward:

```c
void waitInterrupt(int interrupt) {
    Process waiting, p;
    maskInterrupts();
    waiting = removeHead(readyList);
    p = head(readyList);
    ioTransfer(p, interrupt);
    addFirst(readyList, waiting);
    allowInterrupts();
}
```

3.2.3 Idle process
With interrupts, an empty `readyList` is not necessarily an error (if at least one process waits for an interrupt). The simplest way to handle this case is to introduce a process called idle in the kernel: the process simply executes an empty while loop. Therefore, whenever the `readyList` is empty, the kernel executes the idle process. This requires modifying the functions `P()` and `waitInterrupt()`. The idle process can be created within the function `start()`.

3.2.4 Time slicing
We now show how to use `ioTransfer` inside the kernel to implement time slicing. For time slicing, we introduce a “clock” process inside of the kernel. Basically, the clock process, on each clock interrupt, moves the process at the head of the ready list to the rear, and allocates the CPU to the new process at the head of the ready list. If the ready list is empty, the CPU is allocated to the idle process. With `idleProcess` to refer to the idle process, the code of the clock process is as follows:

```c
void clockCode() {
    Process p;
    initClock();
    // initializes the device to generate interrupts
    while (true) {
        p = head(readyList);
        if (p != nil) {
            ioTransfer(p, 0);
            // execute p (0 refers to clock interrupt)
            idleProcess = newProcess(idleCode, idleStackSize);
            ioTransfer(idleProcess, 0);
            p = removeHead(readyList);
            // p is the process interrupted by the clock
            addLast(readyList, p);
            // p is moved to the rear of the ready list
        }
    }
}
```

4. DESCRIPTION OF THE STUDENT PROJECT
Once the students are familiar with the material of Sections 2 and 3, we ask them to implement a kernel different from the one above. The goal is to implement Java monitors [5]. The project is decomposed into two parts, the first not involving interrupts, the second including interrupt handling. We provide to the students the functions `newProcess`, `transfer` and `ioTransfer` introduced in Section 2. Some additional functions are provided, e.g., to initialize the clock, to allow interrupts, to mask interrupts.

4.1 Part 1: Java monitors without interrupt handling
The first version of the kernel has to provide the following interface (parameters to be defined by the students):

```c
createProcess : same as in Section 3.1.1.
start : same as in Section 3.1.1.
createMonitor : Allocates and initializes the data structure for a monitor. Returns the identity of the newly created monitor.
enterMonitor : Called at the beginning of a synchronized method.
wait, notify, notifyAll : Called to execute the corresponding Java methods.
```
exitMonitor : Called at the end of a synchronized method.

Using this first version of the kernel, we asked the students to develop the following small producer/consumer application involving three processes (two producers prod1, prod2, one consumer cons) and a buffer object with (synchronized) methods put(int i) and get(). Process prod1, in an infinite loop, waits until button 1 is pressed (busy waiting) and then calls put(+1). Process prod2, in an infinite loop, waits until button 2 is pressed (busy waiting) and then calls put(-1). Process cons maintains a counter initialized to 0. In an infinite loop, cons calls get(), updates the counter (+1 or -1) depending on the value returned by get(), and displays the new value of the counter using the LEDs.

4.2 Part 2: Adding interrupt handling

The second version of the kernel includes interrupt handling. We ask the students first to add time slicing as described in Section 3.2.4. Second, we ask them to implement the function waitInterrupt described in Section 3.2.1. Third, we ask the students to add the following function to the kernel:

waitDelay (int delay) : same as wait with an additional delay. The process calling this function is unblocked either by the execution of notify, notifyAll, or by expiration of the delay specified.

Finally, we ask the students to write a second small application to test the extended kernel. The application includes two processes, one switching on/off the LEDs of line 1, the other switching on/off the LEDs of line 2. The on/off switching occurs when the corresponding button is pressed (button 1 for line 1, button 2 for line 2), but at least every two seconds. The design of the application is left to the students.

Here is one possible such design, with one producer process prod, two consumers processes cons1, cons2 (the two processes mentioned above), and two buffers of “tokens” buf1 and buf2. The get method of both buffers, if the buffer is empty, calls waitDelay with a delay of 2 seconds. Process prod, in an infinite loop, waits until a button is pressed using waitInterrupt (there is one single interrupt for all 4 buttons). Then, if button 1 is pressed, cons puts a token into buf1; if button 2 is pressed, cons puts a token into buf2. Process cons1, in an infinite loop, calls buf1.get and then switches on/off the LEDs of line 1 (if the LEDs are off, they are switched on; if the LEDs are on, they are switched off). Process cons2 does the same with the LEDs of line 2.

5. PROJECT FEED-BACK

Overall the project reached its goal: A large majority of students found the project interesting, and helpful to better understand Java monitors. Moreover, a large majority of the kernels were implemented in a proper way by the students, and contained minimal mistakes. There were also a small number of projects with serious mistakes. We comment now on these main mistakes. We start with mistakes related to the implementation of the kernel. Later we comment on mistakes related to the application code.

Kernel mistakes. Some students did not implement the right data structures for monitors, namely two process lists, one when entering the monitor was not possible, the other related to wait. Some students introduced only one of these two lists. Some other students had no list at all, using a boolean to know whether the monitor was free. In the latter case, waiting to enter the monitor was done by busy waiting. These errors were surprising, considering that a lecture slide explicitly mentioned these two lists. Some students that correctly introduced these two lists nevertheless showed some basic misunderstandings: processes could be simultaneously in more than one list (including the readyList). Other mistakes were related to the implementation of wait and notifyAll: wait did not release the monitor, and notifyAll was implemented by multiple calls to notify (correct, but inefficient). Some students also implemented unnecessary process switch upon monitor exit: the process exiting the monitor released the CPU. Some implementations did not allow for nested monitor calls (monitor call from within a monitor). Finally, surprisingly, two students tried to implement the monitor kernel on top of the semaphore kernel, i.e., using the operations P and V.

Application mistakes. Because the application was written in C and not in Java, some students provided poorly structured code. Typically, in Java the buffer would be an object with synchronized methods put and get, the first method called by a producer, the second by a consumer. In C, this translates into two functions get and put, each starting with a call to enterMonitor and ending with a call to exitMonitor. Some students did not introduce these two functions: the get code, including enterMonitor and exitMonitor, was directly inserted in the producer code; the put code, including enterMonitor and exitMonitor, was inserted in of the producer code. Some other students did not call enterMonitor and exitMonitor at the right place, i.e., at the beginning and at the end of synchronized methods. Sometimes notify or notifyAll was not used correctly. Finally, while busy waiting was mandatory in part 1 of the project (no interrupt handling) some students still used busy waiting in part 2 to read the buttons.

To summarize, some of these mistakes, both at the kernel level and at the application level, show some fundamental misunderstandings. However, we believe that those students did not invest enough time to try understanding concurrency issues.

6. ACKNOWLEDGMENTS

We would like to thank Paolo Ienne and Xavier Jimenez for their help in using the FPGA board, and Nicolas Schiper for his useful comments on a previous version of the paper.

7. REFERENCES


