# TECHNISCHE UNIVERSITEIT EINDHOVEN
## Faculteit Wiskunde en Informatica

*Examination Operating Systems (2IN05)*
*on September 30, 2008, 9.00h-10.30h.*

The exam consists of two parts that are handed in separately. Part A consists of knowledge-questions that must be answered in maximally 25 minutes, *on a separate sheet.* Extensive explanations are not required in part A, compact answers will be appreciated, answers will be judged correct/wrong only. After you have handed in part A, possibly much sooner than this 25 minutes, you may use the course book and the slides from the lectures for part B. (**Note**: just the lecture slides, no slides concerning exercises or previous exams or any other notes.)

Work clearly. Read the entire parts first before you start. Scores for exercises are indicated between parentheses. The score sums to 10 points. There are 2 pages in total.

---

## PART A (4 points)

---

1. Give at least three motivations for the existence of operating systems.
    a. *abstractie*, of underlying diversity
    b. *virtualisatie*, common concepts (linear memory, file system, processor) available to applications as virtual machine
    c. *sharing,* of functionality needed by all applications
    d. *resource management*
    e. *concurrency,* use resources at the same time
    f. *program portability*, using standardized interfaces

2. Give the steps that make up a conditional critical region.
    a. lock
    b. **while** not condition **do** wait **od**
    c. critical section
    d. possible signals
    e. unlock

3. Give 3 rules of thumb to avoid deadlock in action synchronization.
    a. perform sequences of P operations in fixed order, systemwide;
    b. avoid greedy consumers
    c. let critical sections terminate

4. What are the steps in processing a system call?
   We give it for a *read* system call.
    a. pushing parameters
    b. call library function *read*
    c. put code for *read* in reg.
    d. trap: switch mode and call particular handler
    e. handler calls read function handler

    f. handler performs read actions (store data at address / may need to suspend calling process to perform physical read)
    g. swtich mode back and give control back to caller (return path)

5. What do we mean by *interference* among concurrent processes (or threads)?
    a. *Assumptions or knowledge that one process has about the state are disturbed by actions of another process*

6. When can we regard an assignment in Pascal or C to be atomic, and what is the underlying assumption?
    a. When there is at most 1 reference to a shared variable. Assumption is that interrupts are implemented correctly with respect to internal processor state (registers), and that read and write operations are atomic for the type of that variable.

7. What are the four correctness criteria to take into account in concurrent programs and systems?
    a. functional correctness
    b. absence of deadlock
    c. minimal waiting
    d. fairness

8. Give two motivations for making a program multi-threaded.
    a. follow solution structure: natural concurrency
    b. take advantage of underlying platform concurrency
    c. hide latency

## PART B (6 points)

```
        { Initial state: x = y-1 ^ y = p ^ ¬b}
while ¬b do                              while y ≠ x do
    x := p;            ||                    p := p+1;
    y := p                                   b := x = y
od                                       od
```

1. (2 pt) Given is the above program in which two threads run concurrently using shared variables *x, y* and *p.* All assignments and tests may be regarded as atomic. Give traces to show the following behaviors.
   a) Both threads terminate,
   *(¬b)(y ≠ x)(x:=p)(y:=p)(p:=p+1)( b := x=y)(b)(y=x)*
   b) only the left one terminates,
   *(¬b)(y ≠ x)(x:=p)(p:=p+1)( b := x=y)(y:=p)(b) [(y ≠ x)(p:=p+1)(b := x=y)]∞*
   c) only the right one terminates.
   *(¬b)(x:=p)(y:=p)(y=x) [(¬b)(x:=p)(y:=p)]∞*

2. Given are three threads that use shared variables *x* and *y*.

*int x, y;*

**Proc** *A* = |[ **while** *true* **do** *await(x=y); x := (x+1)* **mod** *N; y := (y+2)* **mod** *N* **od** ]|;
**Proc** *B* = |[ **while** *true* **do** *await(x<y); x := (x+2)* **mod** *N* **od** ]|;
**Proc** *C* = |[ **while** *true* **do** *await(y<x); y := (y+3)* **mod** *N* **od** ]|;

*x, y := 0,0;*
*A || B || C*

*N* is a positive natural number. The *await*-statements indicate that the correspondent assertions are required at those locations in the program text.

1. (2.5) Solve this synchronization problem using Conditional Critical Sections with condition variables, i.e., replace the *await* statements by this CCS implementation. The assignments occurring in the program text are not atomic and need proper protection.
2. (0.5) Explain why deadlock cannot occur for your solution.
3. (1) Is starvation possible? For which value(s) of *N?*

1. There are two possible ways of interpreting the exercise. In the first one the assignments to *x* and *y* are assumed to be the body of the critical section. The second one considers all the second assignment as separate critical section. This allows more concurrency (more traces). We only discuss the first one here as it was chosen by most students. For the second one, notice that *signals* have to occur *inside* critical sections; no concurrent operations on condition variables are allowed.

Introduce three condition variables for the corresponding conditions: *cA, cB* and *cC.* Also one semaphore, *m*, for exclusion, initially 1. We use the standard scheme.

**Proc** *A* = |[ **while** *true*
**do** *P(m);*
**while** $x \neq y$ **do** *Wait (m, cA)* **od;**
*x := (x+1)* **mod** *N; y := (y+2)* **mod** *N;*
*DoSignal;*
*V(m)*
**od** ]|;

**Proc** *B* = |[ **while** *true*
**do** *P(m);*
**while** $x \geq y$ **do** *Wait (m, cB)* **od;**
*x := (x+2)* **mod** *N;*
*DoSignal;*
*V(m)*
**od** ]|;

**Proc** *C* = |[ **while** *true*
**do** **while** $y \geq x$ **do** *Wait (m, cC)* **od;**
*y := (y+3)* **mod** *N;*
*DoSignal;*
*V(m)*
**od** ]|;

Here,

*DoSignal:* **if** $x=y$ **then** *Signal (cA)* **else if** $x<y$ **then** *Signal (cB)* **else** *Signal (cC)* **fi fi**

This can further be optimized for the three cases.

2. Always at least one condition is true, hence, always one thread can make progress. If there is an unjust waiter, *DoSignal* will signal it applying the principle of 'not breaking the chain'.

3. Starvation in this case would mean that one condition never becomes true. For example, if *N=1,* both *x* and *y* remain 0. When *N=3,* the sequence of values *(x,y)* is as follows;
*(0,0)(1,2)(0,2)(2,2)(0,1)(2,1)(2,1)....*
Hence, we have danger of starvation for those values of *N* when there is a repeating sequence that avoids one of the conditions; the start of this sequence must be reachable from the initial state.