TECHNISCHE UNIVERSITEIT EINDHOVEN
Faculteit Wiskunde en Informatica

Examination Operating Systems (2IN05)
on November 3, 2009, 14.00h-15.30h.

The exam consists of two parts that are handed in separately. Part I consists of knowledge-questions that must be answered *on a separate sheet.* Extensive explanations are not required in part I, compact answers will be appreciated, answers will be judged correct/wrong only. After you have handed in part I you may use the course book and the slides from the lectures for part II. (**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. There are 3 pages in total.

PART I (4 points)

---

1. Give at least three motivations for the existence of operating systems.
    a. *Abstraction*: to hide the complexity of the lower levels.
    b. *Virtualization*: to support sharing of computer resources among different users or applications.
    c. *Resource Management*: to address the needs of individual users and their applications while aims at maximizing overall performance of the system.
    d. *Shared functionality*: to provide common functionality (file system, liner memory, APIs) to most programs
    e. *Concurrency*: to enable the simultaneous using of resources.
    f. *Portability*: to support source code portability (unified machine view, standardized on APIs)
2. What are the two principles of condition synchronization?
    a. At places where the truth of a condition is required: *check and block*
    b. At places where a condition may have become true: *signal waiters*
3. What are advantages (at least two) of the use of multiple threads above the use of multiple processes?
    a. Allowing concurrency on the shared memory.
    b. Avoiding context switching overhead.
    c. Improving performance on multiprocessor platforms: multiple processors can run multiple threads.
    d. Hiding latency: while waiting do something useful in another thread.
4. What is the difference between a mode switch and a context switch?
    a. Mode switch: switch between the kernel mode and the user mode; requires very little time.
    b. Context switch: change the memory image; associated with a process switch.
5. What is scheduling, and what is a schedule?
    a. Scheduling is to allocate resources to tasks (processes or threads), which includes decision-making policies to determine the order in which active

processes should compete for the use of resources and the actual binding of a selected task to a resource.

b. Schedule is a function that maps a time and a resource to a task.

6. What is a task attribute, and what is a scheduling metric? Give an example of both.
   a. Attribute is a property a task has. It can be static or dynamic. Examples: arrival time, computation time, required resource, deadline, etc.
   b. A scheduling metric is an observed property. It is the result of applying scheduling policies on a task. Examples: response time, turnaround time, average overshoot, etc.
7. Given are two code fragments of two threads of a program that uses exclusion on variables *x* and *y*.

   ..... *P(m); x := x+y; Sigall (c); V(m)* ....

   ..... *P(m);* **while** $x \leq 0$ **do** *Wait (m, c)* **od ;** *x := x-1; V(m)* **....**

   What are the critical sections of these fragments? (Give traces, or some other clear indication.)

   Critical sections, indicated as (sub)traces:

   a. (*x := x+y); Sigall (c)* { we also accept it without the *Sigall* }
   b. *(x≤0); Wait(m,c)* { we also accept it without the *Wait* }
   c. *(x>0) ; x := x-1;*
8. A spinlock is a common approach to implementing mutual exclusion. What are its advantages and disadvantages, and where is it used?
   a. Advantage: work with multiprocessors.
   b. Disadvantage: busy waiting, cost time, doesn't work for a single processor.

PART II (6 points)

1. (1.5 pt) The following solution to the dining philosophers problem is unfair. This solution was presented during the lecture. Add variables and statements to make it fair. It is not allowed to introduce new semaphores. Operations on indices in the program are modulo *N*.

```
var c: array [0..N) of Condition;
    f: array [0..N) of bool;  { represents whether resource i is free, initially all true }
    m: Semaphore              { initial value 1 }

proc Phil (i: int) =
|[ while true do
    NonCrit (i);

    P(m);
    while not f[i] or not f[i+1] do Wait (m, c[i]) od;
    f[i] := false; f[i+1] := false;
    V(m);

    Crit (i);

    P(m);
    f[i] := true; f[i+1] := true;
    Signal (c[i-1]); Signal (c[i+1]);
    V(m)
  od
]|
```

There are many solutions to this problem. Notice that minimal waiting and fairness cannot be satisfied at the same time (that is, sometimes we must block a *Phil* because there is another *Phil* waiting that takes preference). Therefore, we return to the original solution that had fairness but no minimal waiting.

```
var c: array [0..N) of Condition;
    f: array [0..N) of bool;  { represents whether resource i is free, initially all true }
    m: Semaphore              { initial value 1 }

proc Phil (i: int) =
|[ while true do
    NonCrit (i);

    if i=0
    then  P(m); while not f[i] do Wait (m, c[i]) od; f[i] := false; V(m);      { 1 }
          P(m); while not f[i+1] do Wait (m, c[i]) od; f[i+1] := false; V(m);  { 2 }
    else  P(m); while not f[i+1] do Wait (m, c[i]) od; f[i+1] := false; V(m);  { 3 }
          P(m); while not f[i] do Wait (m, c[i]) od; f[i] := false; V(m);      { 4 }
    fi
```

```
    Crit (i);

    P(m);
    f[i] := true; f[i+1] := true;
    Signal (c[i+1]); Signal (c[i-1])
    V(m)
  od
]|
```

The above solution has two conditional regions after each other. Removing the *V(m); P(m)* (e.g. from end of { 1 } to start of { 2 }) gives a more compact program though it has one interference point less. Note that for i=0, the order must be interchanged to avoid deadlock.

There is one problem remaining with this program: the fairness depends on semaphores (should be strong) and the implementation of *signal*. There are implementations that would allow a *Phil* exit and immediately enter again, overtaking signalled *Phils.*

Most solutions therefore add variables. Solutions that simply limit the number of times one *Phil* may run ahead of others are wrong: they limit progress of *Phil's* even without any competition. Also, if a *Phil* would stop outside its critical section the entire system would stop. This form of non-minimal waiting is erroneous, just like the alternating cars on the bridge example.

For a *Phil* to give precedence to a neighbor at least it must be known that this neighbor is waiting. This can, for example, be done using a counter *x[i]* that denotes the number of times *Phil(i)* waits. *x[i] = 0* indicates that *Phil(i)* is not waiting. Of two waiting neighbors, the one with the larger *x[i]* goes.

The variables *x[i]* are initially 0.

```
proc Phil (i: int) =
|[ while true do
    NonCrit (i);

    P(m);
    while not f[i] or not f[i+1] or x[i] < x[i+1] or x[i] < x[i-1]
    do x[i] := x[i]+1;
       Wait (m, c[i])
    od;
    f[i] := false; f[i+1] := false; x[i] := 0;
    V(m);

    Crit (i);

    P(m);
    f[i] := true; f[i+1] := true;
    Signal (c[i+1]); Signal (c[i-1])
    V(m)
  od
]|
```

This still requires semaphore *m* to be strong, but does not depend on the *signal.* If we want it to work with unfair semaphores we can let a *Phil* adjust the counters of its neighbors rather than its own.

Finally, a solution is to use this counting together with a single condition variable and the *sigall* policy. We leave these last two as an exercise.

2. Given is the following program fragment. There are *N>0 Consumer* threads that run in parallel with 1 *Producer* thread.

var *v: int;*

Proc *Consumer (i: int)* = |[**var** *l: int;* **while** *true* **do** *l := v; "do something with l"* **od**]|;
Proc *Producer* = |[**var** *l: int;* **while** *true* **do** *"compute value in l"; v := l* **od**]|

Synchronize this system using action synchronization (i.e., with semaphores) in the following cases.

a. (0.5 pt) Each value produced by *Producer* is consumed by precisely one *Consumer.*
b. (0.5 pt) The same as a., but now *Producer* may proceed only if a *Consumer* has consumed the value.
c. (1.0 pt) Each Consumer consumes each value.
d. (0.5 pt) The same as c., but now *Producer* may proceed only if each *Consumer* has consumed the value.

Argue the correctness of your solution, for example by giving synchronization conditions. Use as few semaphores as possible (do not introduce more than needed).

a.
var *v: int;*
s, t: Semaphore; { $s_0 = 0; t_0 = 1$ }
Proc *Consumer (i: int)* =
|[**var** *l: int;* **while** *true* **do** *P(s); l := v; V(t); "do something with l"* **od**]|;
Proc *Producer* = |[**var** *l: int;* **while** *true* **do** *"compute value in l"; P(t); v := l; V(s)* **od**]|
This is an example of the bounded buffer problem with N=1.

b.
Two solutions: simple: just add an extra semaphore to the above: *u* with initial value 0.
Proc *Consumer (i: int)* =
|[**var** *l: int;* **while** *true* **do** *P(s); l := v; V(t); V(u) "do something with l"* **od**]|;
Proc *Producer* =
|[**var** *l: int;* **while** *true* **do** *"compute value in l"; P(t); v := l; V(s); P(u)* **od**]|
The addition synchronization follows since $\underline{c}\text{Producerbody} \leq \underline{c}V(u) \leq \underline{c}(\ l := v\ )$

A more advanced solution reorders the first one. Initial value of both semaphores is now 0.
Proc *Consumer (i: int)* =
|[**var** *l: int;* **while** *true* **do** *P(s); l := v; V(t); "do something with l"* **od**]|;
Proc *Producer* = |[**var** *l: int;* **while** *true* **do** *"compute value in l"; v := l; V(s); P(t)* **od**]|
Sychronization conditions of the semaphores and topology properties show
$0 \leq \underline{c}\ (\ v:=l\ ) - \underline{c}(\ l := v\ ) \leq 1$
hence, they alternate.

c.
The easiest solution is to use a pair of semaphores for each consumer. One of these arrays is not strictly needed.
var *v: int;*
s, t[0..N-1]: Semaphore; { $s_0 = N; t_0[0..N-1] = 0$ }
Proc *Consumer (i: int)* =
|[**var** *l: int;* **while** *true* **do** *P(t[i]); l := v; V(s); "do something with l"* **od**]|;
Proc *Producer* =
|[**var** *l: int;* **while** *true* **do** *"compute value in l";*
$P(s)^N$; *v := l; V(t[0]);V(t[1]);…V(t[N-1])*
od]|
Each *Consumer* has its own semaphore *t[i]* which ensures it to read the value of *v* once.

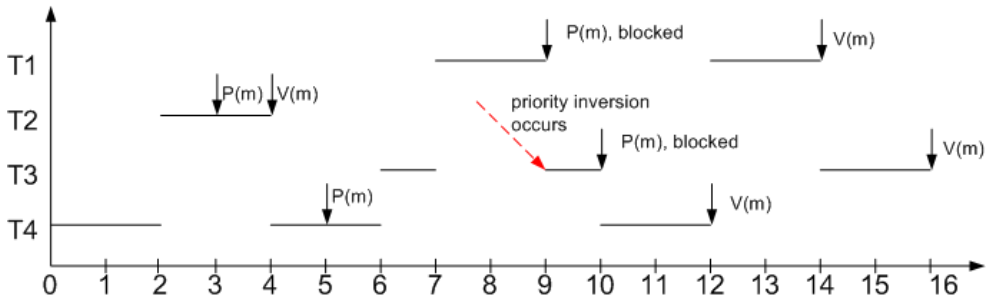
d.
One single extra semaphore as in b. works here as well. But also the reordering.
var *v: int;*
s, t: Semaphore; { $s_0 = N; t_0[0..N-1] = 0$ }
Proc *Consumer (i: int)* =
|[**var** *l: int;* **while** *true* **do** *P(t[i]); l := v; V(s); "do something with l"* **od**]|;
Proc *Producer* =
|[**var** *l: int;* **while** *true* **do** *"compute value in l";*
v := l; V(t[0]);V(t[1]);…V(t[N-1]); $P(s)^N$ **od**]|
Notice that the text `compute ...' should not be inside the critical sections since they represent time consuming computations. For the particular example one may argue that overlapping iterations of the repetition somehow put this action between a *P* and a *V*. However, in general it does not have to be this way.

3. Given are the following four tasks, executed using a pre-emptive scheduler. In this table, a higher priority number means a higher priority for the task. In addition to the processor, the tasks use a single resource that they acquire half-way their execution using a binary semaphore.

Task nr.	Priority	Arrival time	Total service time (computation time)
1	4	7	4
2	3	2	2
3	2	6	4
4	1	0	6

a. (0.5) Draw the resulting schedule.
b. (1.0) Show where priority inversion occurs.
c. (0.5) Suggest a solution for this inversion problem.

a:



b: The priority inversion occurs at time 9 where T1 waits on T4 while the waiting time is increased by T3.

c: Using the priority inheritance protocol dynamically adjusts the priority of T4 to the maximum priority of a waiter: 4. Then T3 will wait.