TECHNISCHE UNIVERSITEIT EINDHOVEN
Faculteit Wiskunde en Informatica

Examination Operating Systems (2R230)
on Tuesday March 20, 2007, 14.00-17.00 uur.

The exam consists of three parts that are handed in separately. Part A consists of knowledge-questions that can certainly be answered in maximally 30 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. When you have handed in part A (possibly sooner than this 30 minutes) you may use books and notes for part B. Part C pertains to either the essay you made or the report delivered for the practical.

Work clearly. Read the entire parts first before you start. Scores for exercises are indicated between parentheses. The score sums to 11 points. There are 3 pages in total.

PART A (3 points)

---

1. Give at least three motivations for the existence of operating systems.
    a. *abstractie*, van onderliggende diversiteit
    b. *virtualisatie*, algemene concepten (lineair geheugen, file systeem, processor) beschikbaar voor applicaties als virtuele machine
    c. *sharing*, van functionaliteit die alle applicaties nodig hebben
    d. *resource management*
    e. *concurrency*, gelijktijdig gebruik kunnen maken van resources
    f. *programma portabiliteit*, d.m.v. gestandaardizeerde interfaces

2. What is the 'blocking factor' in a file system?
    a. The ratio of logical blocks to physical blocks

3. Consider the following statements in a language like C or Pascal:
    x := y;
    x := 5+z;
    Here, *x* and *y* are shared variables and *z* is a local variable (accessible by just one thread). May these statements be regarded as atomic? Motivate your answer in one line.
    a. first assignment contains two references to shared variables: not atomic
    b. second assignment contains just one reference to shared variable: atomic

4. Give two motivations for making a program multi-threaded.
    a. deal with natural concurrency
    b. hide latency
    c. improve performance *on multiprocessor platform.*

5. What is the working set and what determines the lower and the upper limit to its size?

    a. Working set is the set of physical memory pages currently dedicated to a running process. Upper limit to its size: degree of multiprogramming. Lower limit: avoiding occurrence of page faults.

6. Give circumstances when busy waiting is an acceptable synchronization technique. Also indicate when it is not acceptable.
    a. When synchronization between separate processors (physical activities) is needed. It does not make sense to use it when this is not the case, for example, on a single procesor or in most cases at the user program level.

7. In a program with condition variables we use critical sections of the form
   P(m); **while not** *COND* **do** *Wait (m, cv)* **od;** *critical section body*; *Signalling; V(m)*
   with *COND* a boolean expression and *cv* a condition variable. Explain why a repetition is better at this place than a selection (i.e., an **if**-statement).
    a. This reflects the 'signal-and-continue' discipline. It is not guaranteed that the condition holds before a signal. Even then, it is not guaranteed that the resumed process is one blocked on the condition. There might be others waiting on the condition variable or 'newcomers' may overtake. Therefore, the condition must be checked under exclusion.

8. Mention the four correctness concerns in concurrent programs or systems.
    a. functional correctness, minimal waiting, absence of deadlock, fairness

9. Wat is the return path from the kernel and where is it used?
    a. The sequence of checks executed upon resuming user-space execution. It is executed upon returning from a system call or interrupt.

10. Give two machine instructions that can be used for implementing binary semaphores.
    a. Fetch&Add, Test&Set, Compare&Swap

PART B (8 points)

1. Given is the following set of processes, with correspondent maximal numbers of required resources of three possible types. The number of resources for types *r1, r2 and r3* are 3, 3 and 2 respectively. Consider the situation that occurs after execution of
A: Req (r1, 2); A: Acq (r1, 2);

Process	r1	r2	r3
A	3	3	2
B	2	2	2
C	1	1	1

   a) (0.5 pt) Draw the full claim graph for this situation.
   b) (1.5 pt) For each of the following three action sequences, explain if the *Acq's* are admitted according to the bankers algorithm. Draw diagrams, when necessary.
      1. *B: Req (r3, 1); B: Acq (r3, 1)*
      2. *C: Req (r2, 1); C: Acq (r2, 1); B: Req (r2, 1); B: Acq (r2, 1)*
      3. *A: Req (r1, 1); A: Acq (r1, 1); C: Req (r3, 1); C: Acq (r3, 1)*

2. Given is a large set of threads that use a shared bounded buffer. There are many producers and many consumers in this set. The bounded buffer has an array implementation, used in a cyclic fashion. Synchronisation is done through condition variables. The correspondent type definitions are similar to those presented in the slides; variables *cnt* gives the current number of elements in the buffer. In your solutions, you are free to adapt these types as well as the interfaces of the functions below, but you have to use synchronization using condition variables.

```
type buffer =
record  q: queue of elem;
        Prod, Cons: Condition;
        m: Semaphore
end;
```

```
type queue (elem) =
record  b: array [0..N) of elem;
        r, w, cnt: int;
end;
```

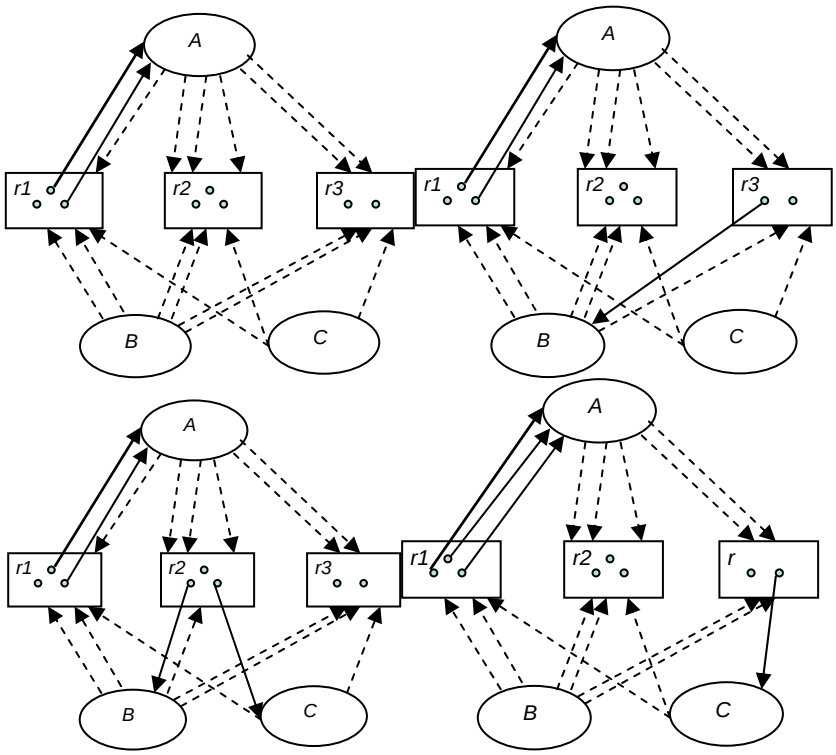
   a) (0.5 pt) Give an implementation of procedures *Send* and *Receive.*
      proc *Send (***var** *b: buffer; x: elem)*
      proc *Receive (***var** *b: buffer;* **var** *y: elem)*
      Use efficient signalling.
   b) (0.5 pt) Give an alternative implementation of procedures *Send* and *Receive* that can deal with multiple elements in a single call; parameter *k* indicates the number of elements.
      proc *Send (***var** *b: buffer; x:* **array of** *elem, k: int /* number of elements */)*
      proc *Receive (***var** *b: buffer;* **var** *y:* **array of** *elem; k: int)*
   c) (1.5 pt) Discuss danger of deadlock, fairness and efficiency of your solution; suggest repairs when needed. Adjust your solution such that calls of *Send* and *Receive* always deal with consecutive elements, even if *k* exceeds *N*. (E.g., a *Send* puts the items in sequence in the buffer without interruption by a call of *Send* by another thread.)

3. (1 pt) In which two cases is the demand paging fetch policy preferable over the prepaging fetch policy?

4. A given file system has the file descriptors contained in directory nodes. The file system is mapped onto a disk of 100000 blocks. A four-byte integer is needed to address each block. A block is 1024 bytes.
   a) (1 pt) Assume a linked-list allocation strategy, where the linked list is stored in the first blocks on the disk (Fig.10-12c in the book). How many blocks are needed for this list? What is the largest possible file that can be represented in this way?
   b) (1.5 pt) Design a solution with the following restrictions. 1) Files shorter than 128 bytes require no additional disk access (besides reading the directory); 2) Blocks of files with a length until 1K can be accessed directly; 3) No restrictions for larger files. Explain what you need in the descriptor and also explain in words how a file is accessed.

SOLUTIONS

Exercise 1.



Solution: the above 4 figures illustrate the solution. The first figure correponds to a). The second figure gives the situation *after* b).1. It is not reducible (neither *A* nor *B* can be reduced) hence the banker's algorithm would forbid the *acq*. For b).2 and b).3 the first *acq's* are possible. The figures give the situations after the seond *acq's*, which are not reducible. Hence, again the banker's algorithm would forbid this.

Exercise 2

a) Follow the standard scheme.

```
proc Send (var b: buffer; x: elem) =
|[ with b, q do
     P(m);
     while cnt = N do Wait (m, Prod) od;
     b[w] := x; w := (w+1) mod N; cnt := cnt+1;
     Signal (Cons);
     V(m)
  od
]|

proc Receive (var b: buffer; var y: elem) =
|[ with b, q do
     P(m);
     while cnt = 0 do Wait (m, Cons) od;
     y := b[r]; r := (r+1) mod N; cnt := cnt-1;
     Signal (Prod);
     V(m)
  od
]|
```

Just a single *Signal* to producer/consumer is enough. A *Sigall* is wakes up too many.

b) A straightforward solution is obtained by just adding a repetition round it. A bit nicer is it when the repetition is brought into the body. I just give the Send here.

```
proc Send (var b: buffer; x: array of elem; k: int) =
|[ var i: int;
   with b, q do
     P(m);
     for i := 0 to k-1 do
       while cnt = N do Wait (m, Prod) od;
       b[w] := x[i]; w := (w+1) mod N; cnt := cnt+1;
       Signal (Cons);
     od;
     V(m)
  od
]|
```

c) There are problems with this solution. It is greedy which can lead to a greediness deadlock. In addition, it uses a Signal/Wait pair merely to count to *k*. In the particular case that the Producer is stuck at a full buffer (or the Consumer at an empty buffer) this is rather inefficient. Finally, it does not guarantee strict sequencing of the items that are sent or received: two different threads may retrieve them in some interleaved fashion (causes perhaps such a greediness deadlock). This interleaving is due only to the unpredictable nature of waking up the *Wait*; in case enough items are available at the start of the *for*, this *for*-loop will be executed without interruption.

The interleaving by itself might be solved by using a semaphore for reservation, though I am not sure whether that can be done (it would give a blocking inside a critical section). Since this does not solve the other problems we do not pursue that here.

A partial solution is to wait until *k* items are available. This makes it impossible to use a single *Signal*; a sequence of *k* signals or easier: a *Sigall* is required.

```
proc Send (var b: buffer; x: array of elem; k: int) =
|[ var i: int;
   with b, q do
     P(m);
     while cnt+k>N do Wait (m, Prod) od;
     for i := 0 to k-1 do b[w] := x[i]; w := (w+1) mod N; cnt := cnt+1 od;
     Sigall (Cons);
     V(m)
  od
]|
```

This solution does not have the mentioned inefficiency: the signalling is done only when the entire work is done. The downside is a lack of fairness. Fairness, however, is difficult to guarantee as it is not known in advance how many items will be produced or consumed.

There is one remaining problem: it deadlocks if *k>N*. In that case the Sending (resp. Receiving) should go in 'batches' of *N* until the remainder is small enough.

```
proc Send (var b: buffer; x: array of elem; k: int) =
|[ var i, batch, done: int;
  with b, q do
     P(m); done := 0;
     while done <> k do
       batch := minimum (k-done, N);
       while cnt+batch>N do Wait (m, Prod) od;
       for i := 0 to batch-1 do b[w] := x[i+done]; w := (w+1) mod N; cnt := cnt+1 od;
       done := done+batch; Sigall (Cons);
     od;
     V(m)
  od
]|
```

Now the deadlock has been removed but the interleaving has come back, at the 'batch'-level. In order to remove this, the first batch must somehow 'reserve' access to the buffer. We introduce an extra variable 'id' for that purpose. It represents the id of the thread that is calling the function. When id=0 the buffer is free for access. This means that the *Send* blocks until there are enough items and the buffer is free for the current thread. We adapt the types and the interface towards this.

```
type buffer =
record  q: queue of elem;
        Prod, Cons: Condition;
        m: Semaphore
        id: int
end;
```

```
type queue (elem) =
record  b: array [0..N) of elem;
        r, w, cnt: int;
end;
```

```
proc Send (var b: buffer; x: array of elem; k: int; thread_id: int) =
|[ var i, batch, done: int;
```

```
  with b, q do
    P(m); done := 0;
    while done <> k do
      batch := minimum (k-done, N);
      while (cnt+batch>N) and (id <>0) and (id <> thread_id) do Wait (m, Prod) od;
      id = thread_id;
      for i := 0 to batch-1 do b[w] := x[i+done]; w := (w+1) mod N; cnt := cnt+1 od;
      done := done+batch; Sigall (Cons);
    od;
    id := 0; V(m)
  od
]|
```

Exercise 3

1) If a process exhibits limited spatial locality.
2) If the cost of a page fault is very low so that preparing does not pay-of in terms of the time saving.

Exercise 4.

a) The linked list takes *k* blocks. Then *k**256 block references are stored to address the remaining 100000-*k* blocks. This means: *k**256 = 100000-*k*, or *k*=100000/257 rounded up = 390 blocks. Largest file: 100000-390 blocks in principle, though place is needed for at least one directory, a filesystem descriptor block (the superblock) and free block information.

b) The descriptor, which is part of the directory structure, must have space to store 128 bytes as well as any required initial references for the remainder. A possible approach is as follows.

- first 128 bytes in the descriptor
- block reference of following 1024 bytes within the descriptor
- start reference of index blocks for following bytes also within the descriptor. This could be the start of a multi-level hierarchy.

In this scheme the descriptor would need 136 bytes for storing the 128 bytes plus the two references. The access procedure for reading would be as follows:

- first 128 bytes: read from descriptor
- use block adress and access next 1024 bytes
- use index hierarchy to access subsequent blocks

A negative point is that this is a fairly complicated scheme that does not align block boundaries very well.