Afdeling Informatica	**Bedrijfssystemen**
Vrije Universiteit	**24.03.2010**

1 Andy Tanenbaum schreef in januari 1992, met als onderwerp 'LINUX is obsolete', het volgende:
"MINIX is a microkernel-based system. [...]. LINUX is a monolithic style system. This is a giant step back into the 1970s. That is like taking an existing, working C program and rewriting it in BASIC. To me, writing a monolithic system in 1991 is a truly poor idea."
Leg uit wat het voornaamste voordeel is van een Operating System dat gebruik maakt van een microkernel, ten op zichte van een die gebruik maakt van een monolithic kernel. *10pt*

2a Leg uit hoe een call naar de library routine `read(fd, &buffer, nbytes);` verwerkt wordt in een monolithic Operating System. Maak gebruik van een plaatje om je antwoord uit te leggen. *5pt*

2b Leg uit hoe een call naar de library routine `read(fd, &buffer, nbytes);` verwerkt wordt in MINIX 3. Maak gebruik van een plaatje om je antwoord uit te leggen. *5pt*

3 De onderstaande implementatie geeft een oplossing van het producer consumer probleem met behulp van binaire semaphoren. Leg in zoveel mogelijk detail uit waarom het producer programma van regel 009 tot en met regel 013 deze volgorde van `up()` en `down()` operaties bevat en nog belangrijker waarom ze in die precieze volgorde staan. *10pt*

```
000 #define N 100
001 int count = 0;
002 semaphore mutex = 1;
003 semaphore full = 0;
004 semaphore empty = 0;
005
006 void producer() {
007     while (TRUE) {
008         item = produce_item();
009         down(&mutex);
010         if (count == N) {
011              up(&mutex);
012             down(&full);
013             down(&mutex);
014         }
015         insert_item(item);
016         count++;
017          if (count == 1) up(&empty);
018         down(&mutex);
019      }
020 }
021
022 void consumer() {
023     while (TRUE) {
024         down(&mutex);
025         if (count == 0) {
026             up(&mutex);
027             down(&empty);
028             down(&mutex);
029         }
030         item = remove_item();
031         count--;
032         if (count == N-1) up(&empty);
033         down(&mutex);
034         consume_item(item);
035     }
036 }
```

4a Wat zijn de vier voorwaarden voor een deadlock? *5pt*

4b Beschouw de onderstaande toekenning van resources R1 t/m R4. Laat zien dat dit een *safe state* is. *5pt*

Process	R1	R2	R3	R4
A	0	1	0	1
B	1	1	0	0
C	0	1	1	0
D	0	0	2	0
E	0	0	2	0

Allocated resources

Process	R1	R2	R3	R4
A	0	0	2	0
B	0	1	0	1
C	1	0	2	0
D	0	3	0	0
E	0	0	1	0

Resources still needed

R1	R2	R3	R4
1	3	6	1

Originally available

5 Leg out hoe het volgende stuk code met slechts een enkele clock meerdere timers simuleert, wees precies. Gebruik een plaatje om je antwoord uit te leggen. *10pt*

```
10494 /*===========================================================================*
10495  *                              do_clocktick                                 *
10496  *===========================================================================*/
10497 PRIVATE int do_clocktick(m_ptr)
10498 message *m_ptr;                          /* pointer to request message */
10499 {
10500 /* Despite its name, this routine is not called on every clock tick. It
10501  * is called on those clock ticks when a lot of work needs to be done.
10502  */
10503
10504   /* A process used up a full quantum. The interrupt handler stored this
10505    * process in 'prev_ptr'.  First make sure that the process is not on the
10506    * scheduling queues.  Then announce the process ready again. Since it has
10507    * no more time left, it gets a new quantum and is inserted at the right
10508    * place in the queues.  As a side-effect a new process will be scheduled.
10509    */
10510   if (prev_ptr->p_ticks_left <= 0 && priv(prev_ptr)->s_flags & PREEM
10511       lock_dequeue(prev_ptr);             /* take it off the queues */
10512       lock_enqueue(prev_ptr);             /* and reinsert it again */
10513   }
10514
10515   /* Check if a clock timer expired and run its watchdog function. */
10516   if (next_timeout <= realtime) {
10517         tmrs_exptimers(&clock_timers, realtime, NULL);
10518         next_timeout = clock_timers == NULL ?
10519                 TMR_NEVER : clock_timers->tmr_exp_time;
10520   }
10521
10522   /* Inhibit sending a reply. */
10523   return(EDONTREPLY);
10524 }
```

6 Beschrijf de structuur van de main loop van een driver process in MINIX 3, wees zo specifiek mogelijk. Leg uit hoe er gebruik gemaakt wordt van message passing en waarom. *5pt*

7a In een embedded computer met slechts 8 MB (8×1024^2 bytes) fysiek geheugen willen we een virtuele geheugenruimte van 32 MB maken, met behulp van paging. Iedere individuele byte in de virtuele geheugenruimte moet te adresseren zijn en de pages worden 1 KB, (1 x 1024) bytes, groot. Daarnaast willen we gebruik maken van een two-level page table, waarvan de second level page tables precies in een page moeten passen. Neem aan dat een page table entry precies 8 bytes omvat.
Geef de layout van een virtueel memory adres, geef uitleg bij je berekeningen. *10pt*

7b Omdat embedded computers niet bijzonder veel intern geheugen hebben zouden we graag wat ruimte besparen en inverted page tables gebruiken. Bereken hoe groot de inverted page table in dit voorbeeld zou zijn, geef uitleg bij je berekeningen. *5pt*

8a Leg uit wat een write-through cache is en geef aan wat het voordeel is van een write-through cache. *5pt*

8b Leg uit hoe MINIX 3 ervoor zorgt dat files consistent blijven, zonder gebruik te maken van een write-through cache. Wat is het voornaamste voordeel van deze aanpak? *5pt*

9 Leg uit hoe *fsck* de staat van een UNIX-achtig file systeem controleert. Leg uit welke tabellen worden opgebouwd en welke waarden aangeven dat er fouten zijn opgetreden. *10pt*

Cijferbepaling: *Het cijfer wordt bepaald door de punten die per onderdeel behaald zijn, bij elkaar op te tellen (totaal maximaal 90 punten), en vervolgens daar 10 extra punten bij te tellen. Er zijn dus totaal 100 punten te behalen. De norm zal worden aangepast wanneer het tentamen te lastig blijkt.*