Faculteit EWI – Afdeling Informatica

Tentamen Besturingssystemen voor INF / TEL (211045),
1 april 2008, 13.30 – 17.00 uur.

Het raadplegen van boeken of diktaten is niet toegestaan.

Invulling van het antwoord op het antwoordformulier is afhankelijk van het aangegeven type vraag:

JNx: Markeer op het antwoordformulier onder JA/NEE VRAGEN bij nummer x het hokje JA of NEE al naar gelang het antwoord ja/wel/juist óf nee/niet/onjuist is.

MKx: Markeer onder MEERKEUZE VRAGEN bij nummer x één van de hokjes A t/m G afhankelijk van de bij de vraag aangegeven alternatieven.

Alle opgaven wegen even zwaar bij de beoordeling (12 punten). De vraagonderdelen per opgave worden meestal ook evenredig gewaardeerd, maar niet altijd.

1. Systeemkenmerken

In de volgende beweringen komt steeds een keuze voor in de vorm [ja-alternatief / nee-alternatief]. Geef aan welk alternatief van toepassing is.

JN1 Het voordeel van multiprogrammering is dat daarmee een [kortere responstijd / hogere throughput] bereikt kan worden.

JN2 Het toepassing van [*multithreading* / *round-robin scheduling*] is typerend voor een timesharing-systeem.

JN3 De term *spooling* slaat op de werkwijze waarbij langzame i/o-operaties via [*DMA*-processoren / *daemon*-processen] op de achtergrond worden uitgevoerd.

JN4 De toegepaste CPU-scheduling in een multi-user-systeem bepaalt in hoge mate de [gebruikers-responstijd / systeem-throughput].

JN5 Een systeem met [*preemptive priority scheduling* / *round-robin scheduling*] is het meest geschikt voor real-time procesbesturing.

JN6 Het interpreteren van gebruikerscommando's is in tegenwoordige besturingssystemen i.h.a. [wel / geen] onderdeel van de kernel.

2. Procesverwerking

Stel bepaalde programmacode wordt uitgevoerd door meerdere threads in een multithreaded programma dan wel door meerdere "klassieke" single-threaded processen. De threads resp. klassieke processen verschillen in de wijze waarop de componenten van de executiecontext wordt gedeeld.

MK1 Welke vorm van *context sharing* is van toepassing bij klassieke *single-threaded* processen. Geef aan welke van onderstaande alternatieven van toepassing is.

MK2 Idem voor *multithreading*.

	MMU-context	*Programmacode*	*Stackpointer*	*Global data*
(A)	shared	non-shared	non-shared	shared
(B)	shared	non-shared	shared	non-shared
(C)	shared	shared	non-shared	shared
(D)	non-shared	shared	shared	non-shared
(E)	non-shared	shared	non-shared	non-shared
(F)	non-shared	shared	non-shared	shared
(G)	non-shared	non-shared	shared	non-shared

De volgende vragen bevatten beweringen over verschillende multithreading-modellen. Geef aan welk alternatief bedoeld wordt.

(A) One-to-One model
(B) Many-to-One model
(C) One-to Many model
(D) Many-to-Many model

MK3 Welk model stelt geen beperking aan het aantal kernel threads?

MK4 Bij welk model zullen alle threads van een proces geblokkeerd raken als één van de threads een blokkerende *system call* uitvoert?

MK5 Welk model vormt een gangbare tussenoplossing waarbij de nadelen van "veel user-threads" tegenover "geen concurrency" beperkt blijven?

MK6 Welk model is als enige van toepassing als een besturingssysteem geen kernel threads ondersteunt.

3. CPU-scheduling

Gegeven zijn jobs die volgens onderstaand schema worden aangeboden voor CPU-verwerking:

	aankomsttijd	*verwerkingstijd*
job1	0	5
job2	2	6
job3	4	4
job4	7	1

Stel deze jobs worden verwerkt volgens onderstaande schedulings-algoritmen:

MK7 Shortest job first

MK8 Shortest remaining time first (preemptive SJF)

MK9 Round-robin met CPU-quantum = 1 (met invoeging van nieuwe jobs vooraan in de ready-queue).

MK10 Round-robin met CPU-quantum = 2 (met invoeging van nieuwe jobs vooraan in de ready-queue).

MK11 Multilevel-feedback-queue-scheduling met queue1 (hoge prioriteit) en queue2 (lage prioriteit). Bij queue1 wordt round-robin toegepast met CPU-quantum = 1 en bij queue2 round-robin met CPU-quantum = 2. Nieuwe jobs komen eerst (vooraan) in queue1 en krijgen maximaal 3 beurten, waarna ze migreren naar queue2 (achteraan).

Nb. Een aankomende job komt onmiddellijk in aanmerking voor scheduling.

Bepaal voor elk algoritme de volgorde waarin de jobs hun verwerking beëindigen:

(A) 1 2 4 3
(B) 1 3 4 2
(C) 1 4 2 3
(D) 1 4 3 2
(E) 3 4 1 2
(F) 4 1 3 2
(G) 4 3 1 2

4. Uitsluiting

Geef bij de volgende onderdelen aan welk antwoord van toepassing is.

MK12 Voor kritieke secties geldt dat:
- (A) binnen de sectie geen interrupts mogen optreden
- (B) wachttijden in geval van blokkering zeer kortdurend moeten zijn
- (C) ze zonder speciale atomaire hardware instructies implementeerbaar zijn
- (D) ze alleen wederzijdse uitsluiting bieden voor uniprocessor-systemen

MK13 De eis van *bounded waiting* die aan een goed lock-unlock protocol wordt gesteld, houdt in dat:
- (A) het aantal processen dat op toegang tot de kritieke sectie wacht, beperkt is
- (B) de beslissing welk proces toegang krijgt niet eindeloos wordt uitgesteld
- (C) de kritieke sectie niet door een proces willekeurig lang bezet mag worden gehouden.
- (D) het aantal keren dat - na het verzoek tot toegang - andere processen voorrang krijgen, begrensd is.

MK14 Zoals bij MK13, maar nu voor de eis van *progress*.

MK15 Geef aan welk programmafragment overeenstemt met een correct lock-protocol gebaseerd op de atomaire *test-and-set*-instructie. Bij deze implementatie wordt gebruik gemaakt van een globale variabele *lock* (met beginwaarde 0) en per proces een locale variabele *key* (met beginwaarde 0).

(A)
```
do { lock = TestAndSet(lock) } while (key);
```
(B)
```
do { key  = TestAndSet(lock) } while (lock);
```
(C)
```
do { key  = TestAndSet(lock) } while (key);
```
(D)
```
do { lock = TestAndSet(key)  } while (lock);
```

MK16 Idem voor een correct lock-protocol gebaseerd op de atomaire swap-instructie.

(A)
```
key = 0;  do { Swap(&key, &lock) } while (!key);
```
(B)
```
key = 1;  do { Swap(&key, &lock) } while (key);
```
(C)
```
key = 1;  do { Swap(&key, &lock) } while (lock);
```
(D)
```
lock = 0; do { Swap(&key, &lock) } while (key);
```

MK17 Instructies om interrupts tijdelijk uit te schakelen vormen een afdoende middel om uitsluiting te verkrijgen:

(A) voor zowel een uniprocessorsysteem als een multiprocessorsysteem
(B) alleen voor een uniprocessorsysteem
(C) alleen voor een multiprocessorsysteem
(D) voor geen van beide

5. Eindige buffersynchronisatie

We beschouwen verschillende implementaties van een eindige FIFO-buffer met functies *produce* en *consume*. De synchronisatie geschiedt met semaforen, waarbij we uitgaan van de gebruikelijke naamgeving en beginwaarden.
Beoordeel de verschillende synchronisatie-varianten op de volgende drie eigenschappen:

- *Concurrency*: de implementatie laat toe dat verschillende bufferplaatsen gelijktijdig kunnen worden gevuld en geleegd
- *Deadlock vrij*: er kan geen deadlock optreden
- *Single blocking*: op elk moment is ten hoogste één producentproces geblokkerd op de semafoor *empty* en ten hoogste één consumentproces op de semafoor *full*

MK18

```
void produce(item *x)
{  wait(empty);
   wait(mutex);
   put_next_in_buffer(x);
   signal(mutex);
   signal(full);
}
```

```
void consume(item *x)
{  wait(full);
   wait(mutex);
   get_next_out_buffer(x);
   signal(mutex);
   signal(empty);
}
```

MK19 [Nb.: t.o.v. MK18 omwisseling volgorde synchronisatie statements]

```
void produce(item *x)
{  wait(mutex);
   wait(empty);
   put_next_in_buffer(x);
   signal(full);
   signal(mutex);
}
```

```
void consume(item *x)
{  wait(mutex);
   wait(full);
   get_next_out_buffer(x);
   signal(empty);
   signal(mutex);
}
```

MK20 [Nb.: t.o.v. MK18 omwisseling alleen bij functie produce]

```
void produce(item *x)
{  wait(mutex);
   wait(empty);
   put_next_in_buffer(x);
   signal(full);
   signal(mutex);
}
```

```
void consume(item *x)
{  wait(full);
   wait(mutex);
   get_next_out_buffer(x);
   signal(mutex);
   signal(empty);
}
```

MK21 [Nb.: t.o.v. MK18 aparte mutex-semaforen voor producenten en consumenten]

```
void produce(item *x)
{  wait(empty);
   wait(prod_mutex);
   put_next_in_buffer(x);
   signal(prod_mutex);
   signal(full);
}
```

```
void consume(item *x)
{  wait(full);
   wait(cons_mutex);
   get_next_out_buffer(x);
   signal(cons_mutex);
   signal(empty);
}
```

MK22 [Nb.: idem t.o.v. MK19]

```
void produce(item *x)
{  wait(prod_mutex);
   wait(empty);
   put_next_in_buffer(x);
   signal(full);
   signal(prod_mutex);
}
```

```
void consume(item *x)
{  wait(cons_mutex);
   wait(full);
   get_next_out_buffer(x);
   signal(empty);
   signal(cons_mutex);
}
```

MK23 [Nb.: idem t.o.v. MK20]

```
void produce(item *x)
{  wait(prod_mutex);
   wait(empty);
   put_next_in_buffer(x);
   signal(full);
   signal(prod_mutex);
}
```

```
void consume(item *x)
{  wait(full);
   wait(cons_mutex);
   get_next_out_buffer(x);
   signal(cons_mutex);
   signal(empty);
}
```

Geef als antwoord het alternatief dat volgens onderstaande tabel de combinatie van eigenschappen aangeeft, die voor de betreffende implementatie geldt:

(A)	*Concurrency*	*Deadlock vrij*	*Single blocking*
(B)	+	+	+
(C)	+	+	−
(D)	+	−	+
(E)	−	+	+
(F)	−	+	−
(G)	−	−	+
(H)	−	−	−

6. Synchronisatieconcepten

In de volgende beweringen komt steeds een keuze voor in de vorm [ja-alternatief / nee-alternatief]. Geef aan welk alternatief van toepassing is.

JN7 Een signal-operatie op een [semafoor / monitorconditie] heeft alleen effect als er processen op de [semafoor / monitorconditie] geblokkeerd zijn.

JN8 Of een proces bij een wait-operatie in een monitor geblokkeerd raakt. hangt [wel / niet] af van de monitorconditie.

JN9 Volgens het *signal-and-wait* principe zal het *signaling process* (dat een signal uitvoert) [eerder / later] binnen de monitor doorgaan dan het *resuming process* (dat doorgestart wordt).

JN10 Als een thread in Java een operatie *x.wait()* uitvoert in een *synchronized method* van een monitor object x wordt de thread geplaatst in de [*entry-set* / *wait-set*] van dit object.

JN11 Het atomair zijn van *concurrent transactions* is gewaarborgd als aangetoond kan worden dat een eventuele *non-serial schedule* door het omwisselen van [conflicterende / niet-conflicterende] read/write-operaties tot een *serial schedule* herleid kan worden.

JN12 Het *two-phase locking protocol* handhaaft *serializability* van transacties doordat het er toe leidt dat soms [read- of write-operaties worden uitgesteld / een transactie wordt afgebroken en na een *roll-back* wordt herstart].

7. Readers & writers monitor

Databestanden zoals files e.d. kunnen zonder bezwaar door meerdere processen simultaan gelezen worden, maar alleen onder uitsluiting beschreven. Deze vorm van synchronisatie staat bekend als het readers & writers synchronisatieprobleem.
Een mogelijke oplossing kan verkregen worden door processen te synchroniseren d.m.v. procedures in een *monitor*. Processen die willen lezen (readers) dan wel schrijven (writers) dienen zich op de volgende wijze vooraf aan te melden en achteraf af te melden:

reader: `StartRead;` "read data only"; `StopRead`

writer: `StartWrite;` "update data"; `StopWrite`

Onderstaand programma beschrijft de monitor die voor de synchronisatie zorgt. De oplossing is zodanig dat writers prioriteit hebben t.o.v. readers: zich aanmeldende readers worden geblokkeerd zolang er writers zijn (bezig of wachtend).

```
monitor ReadersAndWriters {
/* monitor voor readers/writers-synchronisatie
   met prioriteit voor writers */

     int rCount = 0; /* aantal bezige readers */
     int wCount = 0; /* aantal aanwezige writers
                        (bezig of wachtend) */
     int  wBusy = 0; /* writer actief  ? */
     condition readOk, writeOk;

     void StartRead() {
          if (wCount > 0) X1;
          rCount++
          X2;
     }

     void StopRead() {
          rCount--;
          if (rCount == 0) X3;
     }

     void StartWrite() {
          wCount++;
          if (rCount > 0 || wBusy) X4;
          wBusy = 1;
     };

     void StopWrite() {
          wCount--;
          wBusy = 0;
          if (wCount > 0) X5; else X6;
     };
}
```

In de programmatekst stellen X_1 t/m X_6 monitoroperaties weer op de conditievariabelen *readOk* en *writeOk*.

a)

Geef aan welke van de volgende operaties

(A) `readOk.wait();`
(B) `writeOk.wait();`
(C) `readOk.signal();`
(D) `writeOk.signal();`

ingevuld moeten worden voor:

MK24 X1 = ... MK27 X4 = ...

MK25 X2 = ... MK28 X5 = ...

MK26 X3 = ... MK29 X6 = ...

b) Geef een alternatieve implementatie, waarbij de signalering zo efficient mogelijk verloopt en overbodige operaties worden weggelaten (door deze te vervangen door een no-operation: *noop*).

Gevraagd wordt te kiezen uit de de volgende operaties

(A) `readOk.wait();`
(B) `writeOk.wait();`
(C) `readOk.signal();`
(D) `writeOk.signal();`
(E) `readOk.broadcast();`
(F) `writeOk.broadcast();`
(G) `noop;`

en aan te geven welke ingevuld moeten worden voor:

MK30 X1 = ...

MK31 X2 = ...

MK32 X3 = ...

MK33 X4 = ...

MK34 X5 = ...

MK35 X6 = ...

8. Deadlock

a) Beschouw drie processen: P_1, P_2, P_3 en drie resourcetypen: R_1, R_2, R_3 . De beschikbare resources zijn gegeven door de vector: *available* = (1, 2, 2). Gevraagd wordt om voor een gegeven *resource-allocatiegraaf* (RAG) te bepalen of er een *cycle* in voorkomt terwijl er geen *deadlock* is. (Geef antwoord Nee bij deadlock of bij het ontbreken van een cycle.)

Een bepaalde RAG wordt genoteerd als matrix met elementen alsvolgt gedefiniëerd:

$$e_{ij} = \begin{cases} a & \text{als er een } \textit{allocation edge} \text{ bestaat tussen } P_i \text{ en } R_j \\ r & \text{als er een } \textit{request edge} \text{ bestaat tussen } P_i \text{ en } R_j \\ - & \text{als er geen } \textit{edge} \text{ bestaat tussen } P_i \text{ en } R_j \end{cases}$$

JN13

$$\begin{pmatrix} r & a & - \\ r & - & a \\ a & a & a \end{pmatrix}$$

JN15

$$\begin{pmatrix} a & - & a \\ - & a & r \\ r & - & a \end{pmatrix}$$

JN17

$$\begin{pmatrix} a & r & a \\ r & a & - \\ r & a & a \end{pmatrix}$$

JN14

$$\begin{pmatrix} a & a & a \\ r & r & a \\ - & a & r \end{pmatrix}$$

JN16

$$\begin{pmatrix} - & r & a \\ a & a & - \\ r & a & a \end{pmatrix}$$

JN18

$$\begin{pmatrix} a & a & r \\ - & - & a \\ r & a & a \end{pmatrix}$$

b) Stel de maximale resourcebehoefte van elk proces P_i is dezelfde en wordt gegeven door de vector *max* = (1, 1, 1). Gevraagd wordt om voor de eerder gegeven RAG's te bepalen of de toestand volgens het bankiersalgoritme al of niet *safe* is (ja = safe, nee = unsafe).

JN19 RAG zoals gegeven bij JN13

JN20 RAG zoals gegeven bij JN14

JN21 RAG zoals gegeven bij JN15

JN22 RAG zoals gegeven bij JN16

JN23 RAG zoals gegeven bij JN17

JN24 RAG zoals gegeven bij JN18

9. Virtueel geheugen

In de volgende beweringen komt steeds een keuze voor in de vorm [ja-alternatief / nee-alternatief]. Geef aan welk alternatief van toepassing is.

JN25 De grootte van de virtuele adresruimte bij een gegeven processorarchitectuur hangt af van [het aantal bits van programma-adressen / het aantal beschikbare paginaframes].

JN26 Indien bij een virtuele adresafbeelding een [*TLB miss* optreedt / het *invalid bit* in de paginatabelentry aanstaat] zal er een *page-fault-interrupt* gegenereerd worden.

JN27 Meerdere processen kunnen dezelfde programmacode in het hoofdgeheugen adresseren via [noodzakelijkerwijs dezelfde / mogelijk verschillende] virtuele adressen.

JN28 Door gepagineerde segmentatie wordt [externe / interne] geheugenfragmentatie vermeden ten koste van [interne / externe] fragmentatie.

JN29 Bij gebruik van een *hashed page table* wordt het [virtuele paginanummer / paginaframenummer] als *hash value* gebruikt.

JN30 Bij two-level paginering is de paginatabel zelf weer gepagineerd, d.w.z. de paginatabel zal i.h.a. niet volledig [maar wel / en niet] noodzakelijkerwijs consecutief in het geheugen opgeslagen zijn.

10. Gepagineerde segmentatie

Bij het Multics-systeem zijn niet alleen segmenten gepagineerd maar ook de segmenttabel zelf. Een virtueel adres is alsvolgt opgebouwd:

s1	s2	p	d

In de volgende beweringen komt steeds een keuze voor in de vorm [ja-alternatief / nee-alternatief]. Geef aan welk alternatief van toepassing is.

JN31 Het adresdeel [**s1 / s2**] bepaalt welke subpagina van de segmenttabel bij de adresafbeelding van toepassing is.

JN32 De adresdelen [**s1 & s2 / s1 & s2 & p**] worden tezamen gebruikt bij het raadplegen van de TLB.

JN33 Het maximaal aantal segmenten wordt bepaald door het aantal bits van [de adresdelen **s1** en **s2** tezamen / alleen het adresdeel **s2**]

JN34 De maximale aantal verschillende pagina's dat theoretisch geadresseerd kan worden, wordt begrensd door het totaal aantal bits van de adresdelen [**s2 & p / s1 & s2 & p**].

JN35 Het aantal bits van adresdeel **p** bepaalt de maximale grootte van [een segment / de paginatabel].

JN36 Na (succesvolle) TLB-afbeelding [zijn beide adresdelen **p** en **d** / is alleen het adresdeel **d**] nodig om het fysieke geheugenadres te bepalen.

11. Geheugenallocatie

In de volgende beweringen komt steeds een keuze voor in de vorm [ja-alternatief / nee-alternatief]. Geef aan welk alternatief van toepassing is.

JN37 Bij het second-chance algoritme wordt het *reference bit* van een pagina (ook wel als *use bit* aangeduid) bij inspectie op [0 / 1] gezet.

JN38 In uitzonderlijke gevallen kan het voorkomen dat bij [FIFO / LFU] paginavervanging meer paginafouten optreden bij een groter aantal beschikbare paginaframes.

JN39 De pagina's in het geheugen bij locale [FIFO / LRU]-vervanging met n beschikbare frames vormen een deelverzameling van de pagina's bij vervanging met n+1 frames.

JN40 Een nadeel van [globale / locale] paginavervanging is dat processen elkaars voortgang kunnen beïnvloeden.

JN41 De gemiddelde working-set grootte zal toenemen bij keuze van een [kleinere / grotere] window-size.

JN42 Het verschijnsel dat programma's *locality of reference* vertonen heeft tot gevolg dat [*demand-paging* / het *working-set* principe] kan worden toegepast,

12. Paginafouten

Beschouw het volgende pagina-referentiepatroon van een proces (t.a.v. 6 pagina's):

1 2 3 4 2 1 5 2 1 3 6 2 5 2 3 2 1 2 3 5

Gevraagd wordt bij verschillende stategieën te bepalen hoevaak pagina's worden *herladen*.
Elke eerste keer dat een pagina wordt gerefereerd treedt (uiteraard) een paginafout op. Deze paginafouten worden *niet* geteld. Pas wanneer een pagina verwijderd is en daarna weer wordt gerefereerd moet de pagina opnieuw worden geladen en tellen we de paginafout.

MK36 LRU-vervanging als het proces over 4 paginaframes beschikt.

MK37 FIFO-vervanging als het proces over 4 paginaframes beschikt.

MK38 Optimale (OPT-)vervanging als het proces over 4 paginaframes beschikt.

MK39 Working-set algoritme met window-grootte 5 (referenties).

MK40 Working-set algoritme met window-grootte 6 (referenties).

MK41 Bepaal de minimale working-set-grootte die optreedt bij het working-set algoritme met window 6, als we de aanloopperiode (eerste 6 referenties) buiten beschouwing laten.

Mogelijke meerkeuze antwoorden:

(A) =	1	(E) =	5
(B) =	2	(F) =	6
(C) =	3	(G) =	7
(D) =	4		

13. Achtergrondgeheugen

Bij onderstaande formuleringen zijn steeds *twee* keuze-antwoorden van toepassing (juist) en de *twee andere* keuze-antwoorden niet van toepassing (onjuist).
Geef aan welke antwoordparen beide juist en beide onjuist zijn, alsvolgt:

(A) alternatieven 1 + 2 zijn juist en alternatieven 3 + 4 zijn onjuist
(B) alternatieven 1 + 3 zijn juist en alternatieven 2 + 4 zijn onjuist
(C) alternatieven 1 + 4 zijn juist en alternatieven 2 + 3 zijn onjuist
(D) alternatieven 2 + 3 zijn juist en alternatieven 1 + 4 zijn onjuist
(E) alternatieven 2 + 4 zijn juist en alternatieven 1 + 3 zijn onjuist
(F) alternatieven 3 + 4 zijn juist en alternatieven 1 + 2 zijn onjuist

MK42 Als in een directory slechts "non-shared" referenties naar andere directories voorkomen is de directory-structuur mogelijk van het volgende type:
1) Single level
2) Tree-structured
3) Acyclic-graph
4) General-cyclic-graph

MK43 Welke fileallocatie-methoden ondersteunen *direct access* het beste:
1) Contiguous allocation
2) Cluster allocation
3) Linked allocation
4) Index blocks

MK44 Een geschikte methode om de verzameling lege blokken op secundair geheugen bij te houden is:
1) d.m.v.een *bit map* waarbij elk blok door één bit wordt gerepresenteerd
2) door het bijhouden van *reference counts*
3) op dezelfde wijze als waarop de blokken van een file worden geadministreerd
4) d.m.v. een *use-bit* die periodiek gesampled wordt.

MK45 Het toepassen van linked allocation maakt het mogelijk dat:
1) fileblokken in meerdere files kunnen zijn opgenomen
2) extensie van de file eenvoudig is via de pointer naar het laatste blok
3) elk vrij blok op de disk gebruik kan worden om de file uit te breiden
4) *direct-access* naar fileblokken efficient verloopt

MK46 De wijze van fileallocatie in UNIX-systemen is een vorm van:
1) Multilevel allocation
2) FAT-based allocation
3) Linked allocation
4) Indexed allocation

MK47 Stel (tenminste) drie disk i/o requests staan in de wachtrij om uitgevoerd te worden. Bij de volgende schedulings-algoritmen is het mogelijk dat de lees / schrijf-kop voor de afhandeling van elk van de disk-i/o's van richting omkeert.
1) FCFS
2) SSTF
3) SCAN
4) C-SCAN

14. Protectie

Ga uit van het access matrix protectiemodel en beschouw de toegangsrechten ten aanzien van drie protectiedomeinen D_1, D_2 en D_3 en twee files F_1 en F_2, zoals aangegeven in de volgende deelmatrix:

	F_1	F_2	D_1	D_2	D_3
D_1	read			switch	
D_2	owner	read*			control
D_3		write			

Stel een proces P executeert in protectiedomein D_1. Geef aan of proces P al of niet in staat is de aangegeven operatie uit te voeren (eventueel na een aantal tussenstappen).

JN43 Het verlenen van read-recht op de file F_1 aan het protectiedomein D_3.

JN44 Het verlenen van read-recht op de file F_2 aan het protectiedomein D_1.

JN45 Het verlenen van read-copy-recht op de file F_1 aan het protectiedomein D_1.

JN46 Het verlenen van write-recht op de file F_2 aan het protectiedomein D_2.

JN47 Het verwijderen van het read-recht op de file F_2 uit het protectiedomein D_2.

JN48 Het verwijderen van het write-recht op de file F_2 uit het protectiedomein D_3.

JN49 Op éénzelfde moment de files F_1 en F_2 lezen.

JN50 Op éénzelfde moment de file F_1 lezen en de file F_2 schrijven.

JN51 Op éénzelfde moment de file F_1 schrijven en de file F_2 lezen.

JN52 Op éénzelfde moment de files F_1 en F_2 schrijven.

15. UNIX

De volgende vragen hebben betrekking op onderstaande C-programma-fragment:

```
if ( fork() )
       wait(0);
else { execve( ... );
       exit(0);
     }
```

In de volgende beweringen komt steeds een keuze voor in de vorm [ja-alternatief / nee-alternatief]. Geef aan welk alternatief van toepassing is.

JN53 De fork-operatie wordt aangeroepen door [alleen het *parent* proces / zowel het *parent* als het *child* proces].

JN54 De test op de conditie van het if-statement wordt uitgevoerd door [alleen het *parent* proces / zowel het *parent* als het *child* proces].

JN55 De aanroep van het *wait*-statement wordt uitgevoerd door [alleen het *parent* proces / zowel het *parent* als het *child* proces].

JN56 De aanroep van het *execve*-statement wordt uitgevoerd door het [*parent* / *child*] proces.

JN57 Het *parent* en het *child* proces zullen aparte *text*-segmenten hebben als gevolg van de [*fork* / *execve*] operatie.

JN58 Het *parent* en het *child* proces zullen aparte *data*-segmenten hebben als gevolg van de [*fork* / *execve*] operatie.

JN59 De voltooiing van het *wait*-statement vindt plaats [vóór / pas na] de uitvoering van het *exit*-statement.

JN60 Na voltooiing van het *wait*-statement is het [*child* / *parent*] proces geëindigd.

JN61 Een child proces krijgt de status van een *zombie* proces als gevolg van een [*wait* / *exit*] operatie.

Antwoorden tentamen Besturingssystemen 01 april 2008

Beoordeling: cijfer := (score - 30) / 15

Totaal aantal te behalen punten = 180.

Punten zijn over de vraagonderdelen gelijkelijk verdeeld,
tenzij anders aangegeven.

1. **Systeemkenmerken** (12 pt)

JN1	Nee-alternatief
JN2	Nee-alternatief
JN3	Nee -alternatief
JN4	Ja-alternatief
JN5	Ja-alternatief
JN6	Nee-alternatief

2. **Procesverwerking** (12 pt)

MK1	E
MK2	C
MK3	A
MK4	B
MK5	D
MK6	B

3. **CPU-scheduling** (12 pt)

MK7	B	1342 (2 pt)
MK8	D	1432 (3 pt)
MK9	F	4132 (3 pt)

MK10	G	4312 (2 pt)
MK11	F	4132 (2 pt)

4. **Uitsluiting** (12 pt)

MK12	C
MK13	D
MK14	B
MK15	C
MK1	B
MK2	B

5. **Eindige buffersynchronisatie** (12 pt)

MK3	F
MK4	G
MK5	A (= H)
MK6	C
MK7	B
MK8	C

6. **Synchronisatieconcepten** (12 pt)

JN1	Nee-alternatief
JN2	Nee-alternatief
JN3	Nee -alternatief
JN4	Nee-alternatief
JN5	Nee-alternatief

JN6	Ja -alternatief

7. **Readers & writers monitor** (12 pt)

MK9	X1 = A (1 pt)
MK10	X2 = C (1 pt)
MK11	X3 = D (1 pt)
MK12	X4 = B (1 pt)
MK13	X5 = D (1 pt)
MK14	X6 = C (1 pt)
MK15	X1 = A (0 pt)
MK16	X2 = G (2 pt)
MK17	X3 = D (1 pt)
MK18	X4 = B (0 pt)
MK19	X5 = D (1 pt)
MK20	X6 = E (2 pt)

8. **Deadlock** (12 pt:)

JN7	Nee
JN8	Ja
JN9	Nee
JN10	Nee
JN11	Nee
JN12	Ja
JN13	Ja, safe
JN14	Ja, safe
JN15	Ja, safe
JN16	Nee, unsafe
JN17	Nee, unsafe
JN18	Nee, unsafe

9. **Virtueel geheugen** (12 pt:)

JN19	Ja-alternatief
JN20	Nee-alternatief
JN21	Nee-alternatief
JN22	Ja-alternatief
JN23	Ja-alternatief
JN24	Nee-alternatief

10. **Gepagineerde segmentatie** (12 pt:)

JN25	Ja-alternatief
JN26	Nee-alternatief
JN27	Ja-alternatief
JN28	Nee-alternatief
JN29	Nee-alternatief
JN30	Nee-alternatief

11. **Geheugenallocatie** (12 pt)

JN31	Ja-alternatief
JN32	Ja-alternatief
JN33	Nee-alternatief
JN34	Ja-alternatief
JN35	Nee-alternatief
JN36	Nee-alternatief

12. **Paginafouten** (12 pt)

MK21	C (=3)
MK22	D (=4)
MK23	A (=1)
MK24	D (=4)
MK25	C (=3)
MK26	C (=3)

13. **Achtergrondgeheugen** (12 pt)