Tentamen Besturingssystemen (211045),
dinsdag 27 januari 2009, 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 toepassen van [multiprogrammering / timesharing] in besturingssystemen verhoogt de CPU-utilisatie doordat de CPU- en I/O-fasen van programma's elkaar kunnen overlappen.

JN2 Het principe van *spooling* wordt toegepast in timesharing-systemen om de [langzame device I/O / interactieve terminal I/O] te ondersteunen.

JN3 In een batchsysteem zal een hoge CPU-utilisatie leiden tot een [lange *turnaround time* / lage *throughput*].

JN4 DMA-controllers kunnen in een multiprogrammeringssysteem een aanzienlijke bijdrage leveren aan het bereiken van een hoge [CPU-verwerkingscapaciteit / multiprogrammeringsgraad].

JN5 *Swapping* wordt in een multiprogrammeringssysteem bepaald door de [*short-term* / *medium-term*] job scheduler.

JN6 In een real-time systeem wordt een respons binnen een kritische tijd bereikt door toepasssing van [*round-robin* / *priority*] scheduling.

2. Systeemstructuren

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

JN7 *System calls* zijn software-interrupts die vanuit [het besturingssysteem / gebruikersprogramma's] worden gegenereerd.

JN8 Essentiëel onderdeel van *dual mode* CPU-verwerking is dat bij executie van *kernel code* resp. *user code* verschillende [stacks / interruptroutines] gebruikt worden.

JN9 Als in *user mode* gepoogd wordt een geprivilegeerde instructie uit te voeren, wordt deze als illegale instructie beschouwd en als [software interrupt / hardware error] afgehandeld.

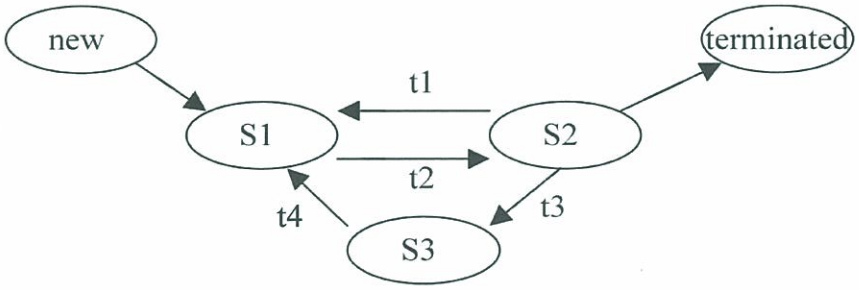
JN10 De *microkernel* benadering houdt in dat de door het systeem geleverde services zoveel als mogelijk in [*user mode* / *system mode*] worden uitgevoerd.

JN11 Systeemprogramma's (zoals bijv */bin/mkdir* in UNIX) verschaffen een gemakkelijke werkomgeving voor gebruikers. Vanuit de kernel gezien wordt [wel / geen] onderscheid gemaakt tussen systeemprogramma's en user-level-programma's.

JN12 Het *virtual machine* concept berust op het principe de hardware van een machine zo te benutten dat [per niveau verschillende / meerdere identieke] virtuele interfaces ontstaan.

3. Procesverwerking

Beschouw het afgebeelde toestandsdiagram van processen en identificeer de juiste beschrijving van de toestanden (S1, S2, S3) en de overgangen (t1, t2, t3, t4).



MK1

(A) S1 = Waiting S2 = Ready S3 = Running
(B) S1 = Waiting S2 = Running S3 = Ready
(C) S1 = Ready S2 = Waiting S3 = Running
(D) S1 = Ready S2 = Running S3 = Waiting
(E) S1 = Running S2 = Waiting S3 = Ready
(F) S1 = Running S2 = Ready S3 = Waiting

MK2

(A) t1: dispatch t2: preempt t3: IO/event completion t4: IO/event wait
(B) t1: dispatch t2: preempt t3: IO/event wait t4: IO/event completion
(C) t1: preempt t2: dispatch t3: IO/event completion t4: IO/event wait
(D) t1: preempt t2: dispatch t3: IO/event wait t4: IO/event completion
(E) t1: IO/event completion t2: IO/event wait t3: preempt t4: dispatch
(F) t1: IO/event wait t2: IO/event completion t3: preempt t4: dispatch

MK3 Bepaal uit onderstaande alternatieven welke context sharing in Unix van toepassing is op een ouder- en zijn kindproces onmiddellijk na de fork-system-call.

(A)	ongedeelde code-section,	gedeelde data-section,	ongedeelde stapel
(B)	ongedeelde code-section,	ongedeelde data-section,	ongedeelde stapel
(C)	gedeelde code-section,	gedeelde data-section,	gedeelde stapel
(D)	gedeelde code-section,	gedeelde data-section,	ongedeelde stapel
(E)	gedeelde code-section,	ongedeelde data-section,	gedeelde stapel
(F)	gedeelde code-section,	ongedeelde data-section,	ongedeelde stapel

MK4 Zoals bij MK3, maar nu voor threads binnen éénzelfde programma.

MK5 Identificeer uit onderstaande alternatieven de meest waarschijnlijke systeemtoestand als de multiprogrammeringsset voornamelijk bestaat uit IO-bound processen.

(A)	weinig preemptie,	lange CPU-bursts,	korte ready-queue
(B)	weinig preemptie,	korte CPU-bursts,	lange ready-queue
(C)	weinig preemptie,	korte CPU-bursts,	korte ready-queue
(D)	veel preemptie,	lange CPU-bursts,	korte ready-queue
(E)	veel preemptie,	korte CPU-bursts,	lange ready-queue
(F)	veel preemptie,	lange CPU-bursts,	lange ready-queue

MK6 Zoals bij MK5, maar nu als de multiprogrammeringsset voornamelijk bestaat uit CPU-bound processen.

4. Processen en threads bij Linux

Dit onderdeel bevat beweringen over het *forken* van processen en het *klonen* van threads onder Linux. Geef aan of de bewering:

(A) alleen juist is voor geforkte processen
(B) alleen juist is voor gekloonde threads
(C) juist is voor beide
(D) onjuist is voor beide

MK7 Na de system call waarbij een proces of thread wordt gecreëerd, wordt de executie van dit proces of deze thread voortgezet bij de functie die meegegeven is.

MK8 Het copy-on-write principe wordt toegepast ten opzichte van het datasegment.

MK9 Gecreëerde processen cq. threads hebben een eigen stack.

MK10 Gecreëerde processen cq. threads hebben een eigen heap.

MK11 Als een kindproces/thread een exit-operatie aanroept, hoeft dit het ouderproces niet te beïnvloeden.

MK12 Mutuele exclusie van een critical region door middel van een mutex-variabele kan in principe correct blijven functioneren, ook als nieuw gecreëerde processen/threads daar aan gaan deelnemen.

5. CPU-scheduling

Geef van de volgende beweringen aan of deze juist of onjuist zijn.

JN13 Een nadeel van *round-robin scheduling* is dat als een job voortijdig eindigt, het resterende deel van een CPU-quantum niet gebruikt wordt, zodat CPU-tijd verloren gaat.

JN14 Bij *preemptive* scheduling zal een proces de CPU vrijgeven als dit proces geblokkeerd raakt, terwijl dit bij *non-preemptive* scheduling niet het geval is.

JN15 Van alle non-preemptieve schedulingdisciplines levert s*hortest-job-first* scheduling altijd de minimale wachttijd op.

JN16 Preëmptieve *SJF scheduling* houdt in dat processen die niet binnen een vaststaand aantal quanta voltooid zijn, worden afgevoerd naar een wachtrij met lagere prioriteit.

JN17 Bij *multi-level-feedback-queue* scheduling kunnen de CPU-quanta die per niveau worden toebedeeld zowel wat betreft grootte als wat betreft aantal verschillen.

JN18 Een probleem van zowel preëmptieve als niet-preëmptieve prioriteitsscheduling is dat het kan leiden tot *starvation*.

6. Uitsluiting

Geef aan welke beweringen gelden voor onderstaande soorten systemen uitgaande van de aanwezigheid van shared-memory.

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

MK13 Instructies om interrupts tijdelijk uit te schakelen vormen een afdoende middel om uitsluiting te verkrijgen.

MK14 Een atomaire test-and-set instructie biedt een afdoende middel om uitsluiting te verkrijgen.

MK15 Spinlocks verschaffen een beter middel voor uitsluiting dan uitschakeling van interrupts.

MK16 In een kritieke sectie afgeschermd door semafooroperaties moeten interrupts worden uitgesloten.

MK17 De gebruik van busy-waiting semaforen (d.w.z. zonder proceswisseling) vormt een acceptabele oplossing.

MK18 Het kritieke-sectie-probleem is in principe op te lossen door middel van software algoritmen, d.w.z. zonder speciale hardware instructies en/of de mogelijkheid om interrupts uit te sluiten.

7. Locking protocol monitor

Om te verzekeren dat transacties volgens een *serializable schedule* worden uitgevoerd worden *locking protocols* toegepast, waarbij data items in *shared-mode* dan wel in *exclusive-mode* gelockt kunnen worden.

Onderstaande programmatekst beschrijft een monitor die de *locking* verzorgt (voor één data item). In de gegeven oplossing wordt prioriteit verleend aan *shared* locking boven *exclusive* locking.

```
monitor LockingProtocol {
/* shared/exclusive locking monitor */

     int shareCount = 0;
     int waitCount = 0;
     int exclusive = 0;
     condition sharedOk, exclusiveOk;

     void SharedLock() {
          waitCount++;
          if (exclusive) X1;
          waitCount--;
          shareCount++;
          if (waitCount > 0) X2;
     }

     void ExclusiveLock() {
          if (shareCount > 0 || exclusive) X3;
          exclusive = 1;
     };

     void Unlock() {
          if (!exclusive) shareCount--;
          exclusive = 0;
          if (shareCount + waitCount > 0) X4; else X5;
     };
}
```

In de programmatekst stellen X_1 t/m X_5 monitoroperaties weer op de conditievariabelen *sharedOk* en *exclusiveOk*.

a) Geef aan welke van de volgende operaties

(A) `sharedOk.wait();` (C) `exclusiveOk.wait();`

(B) `sharedOk.signal();` (D) `exclusiveOk.signal();`

ingevuld moeten worden voor:

MK19 X1 = ...

MK20 X2 = ...

MK21 X3 = ...

MK22 X4 = ...

MK23 X5 = ...

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 volgende operaties

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

en aan te geven welke ingevuld moeten worden voor X_2, X_4 en X_5. Let op: X_1 en X_3 blijven onveranderd.

MK24 X2 = ... MK25 X4 = ... MK26 X5 = ...

8. Monitorimplementaties

Er bestaan verschillende implementatievormen van monitors. We onderscheiden de monitor volgens het "signal and wait" principe zoals gedefineerd door Hoare, de variant hierop van Brinch Hansen (welke is toegepast in Concurrent Pascal) en een *synchronized object* als monitor in Java. Eén van de verschillen betreft de voortgang in de monitor van het *signaling process* (het proces dat de signal-operatie uitvoert of in het geval van Java de notify-operatie) en van het *resuming process* (een wachtend proces dat doorgestart wordt).

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

JN19 Het *signaling process* zal volgens de Hoare-variant [gaan wachten in de *next queue* tot de monitor vrij komt / gaan wachten in een specifieke *condition queue*]

JN20 Het *resuming process* zal volgens de Hoare-variant [de programma-executie *binnen* de monitor continueren / gaan wachten in de *next queue* tot de monitor vrij komt].

JN21 Het *signaling process* zal volgens de variant van Brinch Hansen [de programma-executie *binnen* de monitor continueren / de programma-executie *buiten* de monitor continueren].

JN22 Het *resuming process* zal volgens de variant van Brinch Hansen [de programma-executie *binnen* de monitor continueren / gaan wachten in de *next queue* tot de monitor vrij komt].

JN23 Het *signaling process* in een Java monitor zal [de programma-executie *binnen* de monitor continueren / gaan wachten in de *wait set* tot de monitor vrij is].

JN24 Het *resuming process* in een Java monitor zal [de programma-executie *binnen* de monitor continueren / in de *entry set* geplaatst worden en wachten tot de monitor vrij is].

9. Deadlock

MK27 Welke van de volgende alternatieven is geen correcte methode voor *deadlock prevention*. (Nb. Deadlockpreventie sluit het optreden van deadlocks principieel uit.)

(A) Altijd de maximaal gevraagde resources bij aanvang toe te kennen.

(B) Toegekende resources vrij te geven alvorens nieuwe resources aan te vragen.

(C) Resources in een vaste volgorde te claimen.

(D) Resource-aanvragen te weigeren als een vooraf vastgelegd maximum wordt overschreden.

MK28 Welke bewering t.a.v. een resource-allocatie-graaf (RAG) is onjuist:

(A) De aanwezigheid van een cycle impliceert dat er deadlock is.

(B) Geclaimde resources worden weergegeven door *assignment edges* die zijn gericht van resources naar processen.

(C) In het geval van enkelvoudige resources (*single instance resources*) is de aanwezigheid van een cycle noodzakelijk en voldoende voor het optreden van deadlock.

(D) Bij enkelvoudige resources is een RAG te herleiden tot een corresponderende *wait-for* graaf.

Beschouw 4 processen die eenheden van één type resource claimen. Stel dat deadlock vermeden wordt door het bankiersalgoritme toe te passen. Beoordeel de volgende situatie:

Totaal aantal beschikbare eenheden = **n**

	Proc1	Proc2	Proc3	Proc4
Max. claim	**n**	7	3	5
In gebruik	4	0	2	1

Gevraagd wordt aan te geven welke van de volgende beweringen op de beschreven situatie van toepassing is voor verschillende waarden van **n**.

(A) de situatie kan bij toepassing van het bankiersalgoritme niet voorkomen.

(B) de situatie is safe, maar noch een aanvraag van proc1 voor 1 eenheid, noch een aanvraag van proc2 voor 1 eenheid kan worden gehonoreerd.

(C) alleen een aanvraag van proc1 voor 1 eenheid kan worden gehonoreerd, niet een aanvraag van proc2 voor 1 eenheid.

(D) alleen een aanvraag van proc2 voor 1 eenheid kan worden gehonoreerd, niet een aanvraag van proc1 voor 1 eenheid.

(E) een aanvraag van óf proc1 óf proc 2 elk voor 1 eenheid kan worden gehonoreerd, maar niet voor allebeide samen.

(F) een aanvraag van zowel proc1 en proc 2 elk voor 1 eenheid kan beide worden gehonoreerd.

MK29 Totaal aantal eenheden n = 8

MK30 Totaal aantal eenheden n = 9

MK31 Totaal aantal eenheden n = 10

MK32 Totaal aantal eenheden n = 11

10. 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 virtueel-geheugentechniek maakt het gemakkelijk om [*overlay-programmering* / *swapping*] toe te passen.

JN26 Het aantal pagina's in een virtuele adresruimte is op grond van de gangbare afbeeldingstechniek altijd een macht van twee. Het aantal paginaframes waarin het hoofdgeheugen wordt verdeeld moet [eveneens / niet] noodzakelijkerwijs een macht van twee zijn.

JN27 Een gesegmenteerd virtueel geheugen is vanuit gebruikersoogpunt op te vatten als een grote [1-dimensionale / 2-dimensionale] adresruimte.

JN28 Bij een paginafout wordt [de paginatabel én de TLB bijgewerkt / de paginatabel wel, maar de TLB niet bijgewerkt].

JN29 Bij toepassing van de *inverted page table* techniek is er één paginatabel [voor het gehele systeem / voor elk proces afzonderlijk]

JN30 Na adressering van een gepagineerd segment zullen [wel / niet] alle pagina's van een segment in het geheugen aanwezig zijn.

11. Geheugenbeheer

In onderstaande rijtjes komt steeds één onderdeel voor dat geen relatie heeft met het genoemde onderwerp. Geef aan welk onderdeel dat is.

MK33 Pagina-ophaalstrategie (fetch policy)
- (A) Prepaging
- (B) Multilevel paging
- (C) Demand paging
- (D) Working-set preloading

MK34 Segmentallocatie (placement policy)
- (A) Compaction
- (B) Externe fragmentatie
- (C) First Fit
- (D) Locality model

MK35 Paginavervangingsstrategieën
- (A) Second-chance-algoritme
- (B) LFU-algoritme (Least Frequently Used)
- (C) Best-fit algoritme
- (D) FIFO-algoritme

MK36 Approximatie van LRU-vervanging
- (A) Clock algoritme
- (B) Reference-bit (use-bit) sampling
- (C) Page-fault frequency
- (D) Enhanced second-chance algoritme

MK37 Working-set-model
- (A) Proportional allocation
- (B) Recent paginareferentiepatroon
- (C) Preventie van thrashing
- (D) Locality of reference

MK38 Working-set-grootte
- (A) Pagefault rate
- (B) MFU-algoritme (Most Frequently Used)
- (C) Page reference string
- (D) Working-set window

12. Paginafouten

Beschouw het volgende pagina-referentiepatroon van een proces (t.a.v. 7 pagina's)

1 2 3 4 2 1 5 6 4 1 2 1 2 7 1 6 5 3 2 5 6 1

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.

MK39 LRU-vervanging als het proces over 5 paginaframes beschikt.

MK40 FIFO-vervanging als het proces over 5 paginaframes beschikt.

MK41 Optimale (OPT-)vervanging als het proces over 5 paginaframes beschikt.

MK42 Working-set algoritme met window-grootte 5 (referenties).

MK43 Working-set algoritme met window-grootte 6 (referenties).

MK44 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
(B) = 2
(C) = 3
(D) = 4
(E) = 5
(F) = 6
(G) = 7

13. Filesystemen

Geef van de volgende beweringen aan of deze juist of onjuist zijn.

JN31 De *file allocation table* (FAT) wordt gebruikt bij filesystemen met *indexed allocation* om vast te leggen hoe opeenvolgende fileblokken geketend zijn.

JN32 In een acyclische directory structuur is het bijhouden van een *reference count* afdoende om te weten of een file nog toegankelijk is dan wel of een file verwijderd kan worden.

JN33 In een multi-user-systeem zullen gewone user-files die door één proces zijn geopend, in een *user-open-file-table* worden bijgehouden, en systeemfiles die door meerdere processen kunnen zijn geopend, in een *system-open-file-table*.

JN34 Het invoeren van *links* in filedirectories is een mogelijke techniek om de beperkingen van een puur hiërarchische toegangsstructuur te omzeilen.

JN35 Een acyclische directorystructuur blijft gewaarborgd als er, zoals in UNIX-systemen gangbaar is, alleen meerdere *hard links* naar gewone files mogen voorkomen en geen meervoudige *hard links* naar subdirectories.

JN36 *Indexed allocation* is wat betreft disk i/o scheduling gunstiger dan *linked allocation* omdat in het eerste geval fileblokken een vaste plaats hebben en in het laatste geval verspreid over de disk liggen.

14. Protectie

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

MK45 Welk onderdelen hebben betrekking op een hiërarchische vorm van protectie ?

1. access list
2. dual-mode operation
3. lock-key mechanisme
4. access brackets zoals bij Multics

MK46 Wat is een correcte interpretatie van het access matrix model ?

1. Een rij in de access matrix is te beschouwen als de capability list van een bepaald protectiedomein.
2. De kolommen in de access matrix geven de rechten aan die gelden op verschillende protectieniveau's.
3. Elk access matrix element beschrijft de objecten die - gegeven een bepaalde combinatie van protectierechten - toegankelijk zijn.
4. Elk protectiedomein heeft zowel een eigen rij als een eigen kolom, zodat het "recht van overgang" tussen domeinen in de matrix kan worden vastgelegd.

MK47 Het read* toegangsrecht mag in een access-matrix-entry worden toegevoegd op grond van ?

1. switch right
2. owner right
3. limited copy right
4. transfer right

MK48 Onder welke voorwaarde mag er een toegangsrecht worden verwijderd uit de access-matrix-entry `access(Di,Dj)` behorend bij twee verschillende protectiedomeinen Di en Dj door een proces in een derde protectiedomein Dk?

1. *owner right* $\in$ `access(Di,Dk)`
2. *owner right* $\in$ `access(Dk,Dj)`
3. *control right* $\in$ `access(Dk,Di)`
4. *control right* $\in$ `access(Dk,Dj)`

MK49 In welke gevallen is protectie gebaseerd op capability lists efficienter dan protectie met access lists ?

1. Een file wordt veelvuldig gelezen.
2. Een proces wisselt tijdelijk van protectiedomein.
3. De eigenaar van file X geeft toegangsrechten op deze file aan gebruiker Y.
4. De eigenaar van een file trekt alle toegangsrechten voor andere gebruikers in.

MK50 Wat geldt voor protectie in UNIX systemen ?

1. Protectierechten zijn gekoppeld aan de *user-id's* van processen, d.w.z. elke *user* heeft in principe een eigen protectiedomein
2. Fileprotectie is gebaseerd op *access lists*, d.w.z. de toegangsinformatie is aan het object gekoppeld.
3. De protectie is hiërarchisch georganiseerd zodat de privileges die gelden voor de drie categorieën (*user*, *group* en *others*) genest zijn.
4. Door het zetten van het *setuid-bit* kunnen verleende toegangsrechten worden herroepen

15. UNIX

Geef van de volgende beweringen aan of deze juist of onjuist zijn.

JN37 Als in UNIX een programma middels een shellcommando wordt opgestart, zal het shell-proces een *fork* uitvoeren waardoor twee shell-processen ontstaan, waarvan één vervolgens het programma gaat uitvoeren.

JN38 Veelgebruikte shellcommando's zoals bijv. *ls* en *cat* worden vanuit de shell direct d.m.v. system calls uitgevoerd en leiden niet tot het opstarten van een apart programma.

JN39 Bij een fork-system-call hoeft voor het kindproces geen nieuwe text-structure aangemaakt te worden omdat de bestaande text-structure van het ouderproces - onder aanpassing van de use-counter - gedeeld kan worden

JN40 Indien in UNIX meerdere signals naar een proces verstuurd worden, kunnen signals van hetzelfde type overschreven worden en kunnen signals van verschillend type in willekeurige volgorde aankomen.

JN41 Omdat meerdere processen in UNIX dezelfde file kunnen openen, wordt de *offset pointer* voor lezen of schrijven in de inode van een file bijgehouden.

JN42 Een programma in Unix draait boven op de user interface dan wel boven op de programmer interface afhankelijk van het feit of het in user-mode of in system-mode runt.