Tentamen Besturingssystemen voor INF (211045),
dinsdag 7 april 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 Multiprogrammering beoogt de CPU-utilisatie te vergroten door CPU- en I/O-verwerking [binnen programma's zelf / van programma's onderling] te laten overlappen.

JN2 De multiprogrammerings-set in een batchsysteem wordt bepaald door de toegepaste [CPU-scheduling / medium-term scheduling].

JN3 In een batchsysteem kan de *turnaround-time* van een job [wel / niet] beïnvloed worden door het aanbod en de verwerkingstijden van andere jobs.

JN4 De combinatie van multiprogrammering en [*spooling / round-robin scheduling*] is typerend voor time-sharing systemen.

JN5 In een multiprogrammeringssysteem met *spooling* zullen [DMA-controllers / systeemprocessen] de spooling verzorgen.

JN6 Een systeem met preemptieve [*priority / round-robin*] scheduling is typisch geschikt voor *soft-real-time* toepassingen.

2. CPU-verwerking

Van de processen in de *multiprogramming set* zullen sommige *runnable* zijn (in toestand *running* of *ready*) en sommige *non-runnable* (in toestand *waiting* of *blocked*) Het aantal *runnable* en *non-runnable* processen zal onder invloed van procesacties of systeemgebeurtenissen voortdurend veranderen.

Geef voor de genoemde acties of gebeurtenissen aan hoe het aantal *runnable* processen en het aantal *non-runnable* processen verandert, nl. of het toeneemt (++), gelijk blijft (==) of afneemt (--).

Mogelijke alternatieven zijn:

	# runnable	*# non-runnable*
(A)	==	==
(B)	++	==
(C)	==	++
(D)	--	==
(E)	==	--
(F)	--	++
(G)	++	--

MK1 Fork system call

MK2 Synchronous I/O request

MK3 Time slice expired

MK4 Page fault

MK5 I/O completion interrupt

MK6 Swap-out of a sleeping process

3. Multithreading

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

JN7 Het *copy-on-write* principe van het datasegment wordt [wel / niet] toegepast bij *multithreaded* programma's.

JN8 Threads gebruiken afzonderlijke stackruimten. Dit geldt noodzakelijkerwijs [ook / niet] voor de heapruimte.

JN9 Het [*one-to-one / one-to-many*] multithreading-model stelt geen beperking aan het aantal kernel threads.

JN10 Het [*many-to-one / one-to-many*] model is als enige van toepassing als een besturingssysteem geen *kernel threads* ondersteunt.

JN11 Multithreading op basis van het [*many-to-many / one-to-many*] model vormt een gangbare tussenoplossing waarbij de nadelen van "veel user-threads" tegenover "geen concurrency" beperkt blijven.

JN12 Mutuele exclusie op basis van mutex-variabelen blijft [wel / niet] behouden als threads binnen een *critical region* worden gecreëerd.

4. CPU-scheduling

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

	aankomsttijd	*verwerkingstijd*
job1	0	4
job2	1	5
job3	3	3
job4	5	1

Stel deze jobs worden verwerkt volgens onderstaande schedulings-algoritmen:

(A) Shortest job first

(B) Shortest remaining time first

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

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

NB. Een nieuwe job komt onmiddellijk voor CPU-scheduling in aanmerking.

MK7 Welk schedulingalgoritme veroorzaakt de kleinste gemiddelde wachttijd?

MK8 Welke schedulingalgoritme veroorzaakt de grootste gemiddelde wachttijd?

MK9 Bij welk algoritme heeft job3 de langste turnaroundtijd?

MK10 Bij welk algoritme eindigt de verwerking van job3 eerder dan van job4?

MK11 Bij welk algoritme eindigt de verwerking van job3 eerder dan van job1?

MK12 Bij welk algoritme is er geen preemptie van job1 en wel van job3?

5. Atomic transactions

Stel de volgende read / write transacties T_0, T_1, T_2 op gemeenschappelijk datastructuren A, B, C worden concurrent uitgevoerd.

T_0: read (C); read (A);

T_1: read(A); write (B); write(C);

T_2: read (C); read (B); write(C);

Geef voor onderstaande, verschillende concurrente realisaties aan of deze wel of niet *conflict serializable* zijn.

JN13

T_0	T_1	T_2
read(C)		
	read(A)	
		read(C)
read(A)		
		read(B)
	write(B)	
	write(C)	
		write(C)

JN15

T_0	T_1	T_2
		read(C)
	read(A)	
		read(B)
read(C)		
		write(C)
	write(B)	
read(A)		
	write(C)	

JN14

T_0	T_1	T_2
read(C)		
	read(A)	
		read(C)
read(A)		
		read(B)
		write(C)
	write(B)	
	write(C)	

JN16

T_0	T_1	T_2
		read(C)
	read(A)	
read(C)		
read(A)		
	write(B)	
	write(C)	
		read(B)
		write(C)

Geef nu voor de verschillende concurrente realisaties aan of deze bij toepassing van het *two-phase locking protocol* wel of niet kunnen optreden.

JN17 Realisatie zoals bij JN13.

JN18 Realisatie zoals bij JN14.

JN19 Realisatie zoals bij JN15.

JN20 Realisatie zoals bij JN16.

6. Semafoorsynchronisatie

Beschouw de volgende gevallen (A) t/m (G) van paarsgewijze programmafragmenten met een aanroep van de functie X resp. Y. De aanroepen van X en Y worden onderling gesynchroniseerd door semafooroperaties P (= *wait*) en V (= *signal*).
De gebruikte semaforen hebben de volgende beginwaarden: *s0* (**init** 0), *t1* (**init** 1), *u1* (**init** 1), *v1* (**init** 1), *w2* (**init** 2), de variabelen *c1* en *c2* hebben beginwaarde 0.

(A)

```
P(w2);  |  P(s0);
X;      |  Y;
V(s0);  |  V(w2);
```

(B)

```
P(t1);  |  P(u1);
X;      |  Y;
V(t1);  |  V(u1);
```

(C)

```
P(t1);  |  P(s0);
X;      |  Y;
V(s0);  |  V(t1);
```

(D)

```
P(w2);  |  P(v1);
X;      |  Y;
V(v1);  |  V(w2);
```

(E)

```
P(w2);  |  P(s0);
P(t1);  |  P(u1);
X;      |  Y;
V(t1);  |  V(u1);
V(s0);  |  V(w2);
```

(F)

```
P(t1);                  |  P(u1);
if (c1 == 0) P(v1);     |  if (c2 == 0) P(v1);
c1++;                   |  c2++;
V(t1);                  |  V(u1);
X;                      |  Y;
P(t1);                  |  P(u1)
c1--;                   |  c2--;
if (c1 == 0) V(v1);     |  if (c2 == 0) V(v1);
V(t1);                  |  V(u1);
```

(G)

```
P(t1);
if (c1 == 0) P(v1);     |
c1++;                   |
V(t1);                  |  P(v1);
X;                      |  Y;
P(t1);                  |  V(v1);
c1--;                   |
if (c1 == 0) V(v1);     |
V(t1);                  |
```

Stel dat de codefragmenten door meerdere processen potentieel concurrent worden uitgevoerd. We gebruiken de notatie || om onderlinge *concurrency* aan te geven, d.w.z zonder nadere synchronisatie geldt X || X, X || Y en Y || Y. Als de functies X en Y volgens de codefragmenten worden gesynchroniseerd, zal de concurrency eventueel inperkt zijn zoals in onderstaande meerkeuzevragen is omschreven. De notatie ↔ betekent hierbij *niet-concurrent*, d.w.z. met X ↔ Y wordt aangegeven dat uitvoeringen van X en Y elkaar uitsluiten. Geef aan welk paar codefragmenten de aangegeven synchronisatie oplevert.

MK13 De functies X en Y worden nooit samen uitgevoerd. d.w.z. er geldt X ↔ Y, maar verder wel X || X en Y || Y.

MK14 De procedures X en Y worden om en om afwisselend uitgevoerd. (Er geldt bijgevolg X ↔ X, X ↔ Y en Y ↔ Y.)

MK15 Ten hoogste één uitvoering van X en ten hoogste één uitvoering van Y wordt tegelijktijd uitgevoerd. (Er geldt dus X ↔ X, Y ↔ Y en X || Y.)

MK16 Readers-writers-synchronisatie met X = read en Y = write.

MK17 Eindige-buffer-synchronisatie met X = put en Y = get waarbij "dubbele buffering" wordt toegepast en we toestaan dat X || X en Y || Y.

MK18 Eindige-buffer-synchronisatie met X = put en Y = get waarbij "dubbele buffering" wordt toegepast en we stellen dat X ↔ X en Y ↔ Y.

7. Monitorsynchronisatie

Beschouw een monitoroplossing voor synchrone communicatie tussen meerdere producent en meerdere consument processen via een gemeenschappelijke pipe. Producentprocessen roepen de monitorprocedure *put*() aan en consumentprocessen de monitorprocedure *get*(). Steeds zal één put-operatie synchroniseren met één get-operatie (= rendez-vous) en na overdracht van het betreffende item zullen beide processen de operatie voltooien. Producentprocessen zullen soms wachten op consumentprocessen of consumentprocessen zullen soms wachten op producentprocessen maar nooit wachten beide typen processen tegelijkertijd.
Onderstaand programma beschrijft de monitor die voor de synchronisatie zorgt.
Nb. We gaan er van uit dat de monitorsynchronisatie werkt volgens het Hoare-principe, d.w.z. op basis van *signal-and-wait*.

```
monitor SyncPipe
{
  item buf;
  int  wPuts = 0; /* aantal wachtende producenten */
  int  wGets = 0; /* aantal wachtende consumenten */
  condition putOk, getOk;
  void Put (item *x) {
    if (wGets > 0) {
      wGets--;
      memcpy(&buf ,x); /* A */
      X1
    } else {
      wPuts++;
      X2;
      memcpy(&buf, x); /* B */
    }
  }
  void Get (item *x) {
    if (wPuts > 0) {
      wPuts--;
      X3;
      memcpy(x, &buf); /* C */
    } else {
      wGets++;
      X4;
      memcpy(x, &buf); /* D */
    }
}
```

In de programmatekst stellen X_1 t/m X_4 monitoroperaties weer op de conditievariabelen *putOk* en *getOk*.

Geef aan welke van de volgende operaties

(A) getOk.wait()
(B) putOk.wait()
(C) getOk.signal()
(D) putOk.signal()

ingevuld moeten worden voor:

MK19 X1 = ... MK20 X2 = ... MK21 X3 = ... MK22 X4 = ...

Geef aan welk van de statements `memcpy(&buf, x)` of `memcpy(x, &buf)` (in de programmatekst gemarkeerd met het te kiezen alternatief A, B, C of D) in de tijd gezien als eerste in de monitor wordt uitgevoerd nadat de monitoroperatie Xi (voor i = 1, 2, 3, 4) is aangeroepen.

MK23 X1 = ... MK24 X2 = ... MK25 X3 = ... MK26 X4 = ...

8. Deadlock

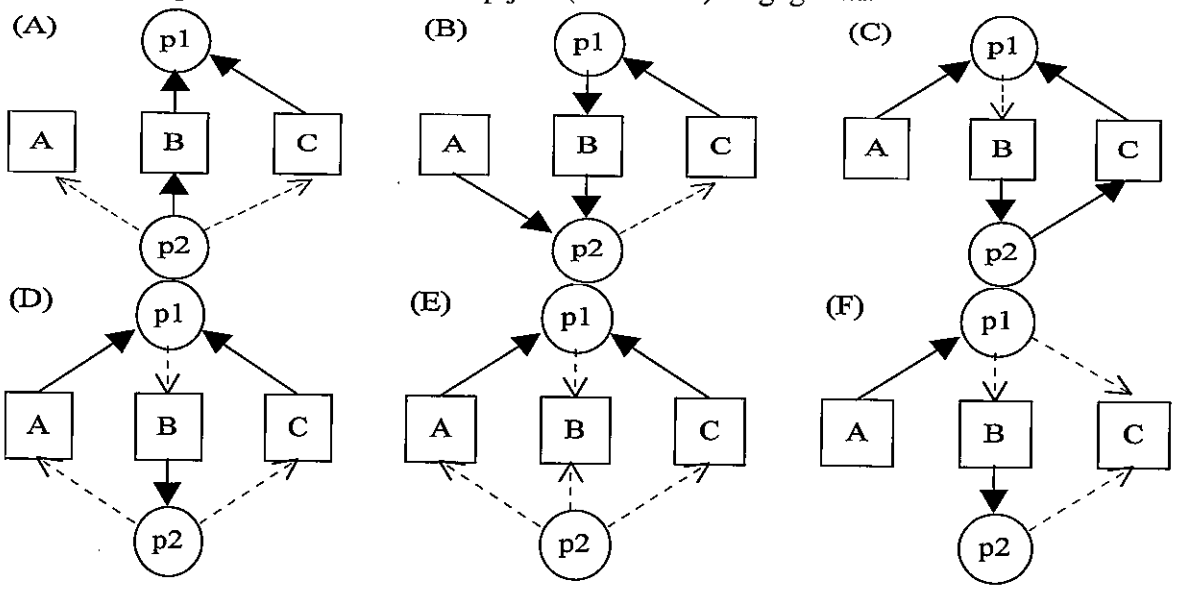
Processen p1 en p2 doorlopen kritieke secties A, B en C volgens onderstaand d.m.v. P- en V-operaties aangegeven schema (A, B en C zijn de bijbehorende exclusie semaforen.)

proces p1: P(A); . . . P(C); . . . V(A); . . . P(B); . . . V(B); . . . V(C)
proces p2: P(B); . . . P(A); . . . V(A); . . . P(C); . . . V(B); . . . V(C)

We beschouwen een zestal situaties waarbij de processen steeds in verschillende toestanden van verwerking verkeren. De actuele positie van de *instruction counter* wordt aangegeven door **ICi↓** of door **ICi→** indien een proces geblokkeerd is op de direct volgende P-operatie.

MK27	proces p1:	P(A); P(C); **IC1↓** V(A); P(B); V(B); V(C)
	proces p2:	P(B); P(A); V(A); **IC2→**P(C); V(B); V(C)
MK28	proces p1:	P(A); **IC1↓** P(C); V(A); P(B); V(B); V(C)
	proces p2:	P(B); P(A); V(A); **IC2↓** P(C); V(B); V(C)
MK29	proces p1:	P(A); P(C); **IC1↓** V(A); P(B); V(B); V(C)
	proces p2:	P(B); **IC2↓** P(A); V(A); P(C); V(B); V(C)
MK30	proces p1:	P(A); P(C); V(A); **IC1→**P(B); V(B); V(C)
	proces p2:	P(B); P(A); **IC2↓** V(A); P(C); V(B); V(C)
MK31	proces p1:	P(A); P(C); V(A); P(B); **IC1↓** V(B); V(C)
	proces p2:	**IC2→**P(B); P(A); V(A); P(C); V(B); V(C)
MK32	proces p1:	P(A); P(C); **IC1↓** V(A); P(B); V(B); V(C)
	proces p2:	**IC2↓** P(B); P(A); V(A); P(C); V(B); V(C)

Geef voor elk van de situaties aan welke van onderstaande RAG's van toepassing is. In deze RAG's zijn *claim edges* d.m.v. onderbroken pijlen ( )aangegeven.



Geef tevens voor elke situatie aan of deadlock nog te vermijden is door een proces ook op een P-operatie te blokkeren als de toestand volgens het bankiersalgoritme onveilig zou worden (antwoord: Ja) of dat deadlock niet meer te vermijden is (antwoord: Nee).

JN21 Situatie zoals bij MK27.

JN22 Situatie zoals bij MK28

JN23 Situatie zoals bij MK29

JN24 Situatie zoals bij MK30

JN25 Situatie zoals bij.MK31

JN26 Situatie zoals bij.MK32

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.

JN27 Het verschil tussen het aantal bits van een virtueel en een fysisch adres is [afhankelijk / onafhankelijk] van de gehanteerde paginagrootte.

JN28 Als bij adresophoging een paginagrens wordt overschreden, wordt [een volgende pagina geadresseerd / automatisch een paginafout gegenereerd].

JN29 Bij een paginafout zal [zowel de betreffende paginatabel als de *TLB* / alleen de paginatabel] worden bijgewerkt.

JN30 Bij two-level paginering zal de *translation look-aside buffer* [slechts één keer / tweemaal] geraadpleegd worden.

JN31 Als meerdere processen eenzelfde paginaframe adresseren via verschillende virtuele adressen is er sprake van [*overlay programmering* / *page sharing*].

JN32 Bij toepassing van de *inverted page table* techniek is er één paginatabel [voor het gehele systeem / voor elk proces afzonderlijk]

10. Geheugenbeheer

JN33 Een nadeel van niet-gepagineerde geheugentoewijzing is het feit dat het geheugen soms niet volledig benut kan worden vanwege [interne / externe] fragmentatie.

JN34 In uitzonderlijke gevallen kan het voorkomen dat bij [FIFO / LFU] paginavervanging meer paginafouten optreden bij een groter aantal beschikbare paginaframes.

JN35 Bij het second-chance algoritme wordt het *reference bit* van een pagina (ook wel als *use bit* aangeduid) bij inspectie op [0 / 1] gezet.

JN36 Toepassing van het working-set strategie draagt ertoe bij dat [thrashing / geheugenfragmentatie] wordt voorkomen:

JN37 De gemiddelde *working set* grootte zal [toenemen / afnemen] bij keuze van een grotere *window size*

JN38 Het [*buddy* / *second chance*] algoritme is een benaderende implementatie van het LRU-paginavervangingsalgoritme.

11. Localiteit

Geef van de volgende beweringen aan of deze een gevolg zijn van het verschijnsel dat bekend staat als *localiteit van programmareferenties* (antwoord = ja) of dat ze hiervan onafhankelijk zijn (antwoord = nee).

JN39 Bij *demand paging* worden pagina's niet eerder in het geheugen geladen dan dat ze geadresseerd worden.

JN40 De pagina's in het geheugen bij LRU-vervanging met n beschikbare frames vormen een deelverzameling van de pagina's bij LRU-vervanging met n+1 frames.

JN41 Het kan voordelig zijn bij hervatting van een proces tot *prepaging* van de working-set over te gaan.

JN42 Toepassing van de working-set strategie is een bruikbare methode om de *page-fault-rate* te beperken en thrashing te voorkomen.

JN43 De interne fragmentatie vermindert bij kleinere paginagrootte.

JN44 Een kleinere paginagrootte resulteert in een betere utilisatie van het geheugen.

12. Paginafouten

Beschouw het volgende pagina-referentiepatroon van een proces (t.a.v. 5 pagina's):

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

Bij de laatste referentie van deze string (naar pagina 5) kunnen zich verschillende gevallen voordoen afhankelijk van het gevolgde pagineringsalgoritme. Geef aan welk van de onderstaande gevallen A t/m G van toepassing is voor elk der daarna genoemde geheugenstrategiën.

(A) er treedt geen paginafout op en er wordt geen pagina verwijderd

(B) er treedt geen paginafout op en er wordt wel een pagina verwijderd

(C) er treedt wel een paginafout op en er wordt geen pagina verwijderd

(D) er treedt wel een paginafout op en pagina 1 wordt verwijderd

(E) er treedt wel een paginafout op en pagina 2 wordt verwijderd

(F) er treedt wel een paginafout op en pagina 3 wordt verwijderd

(G) er treedt wel een paginafout op en pagina 4 wordt verwijderd

MK33 FIFO-vervanging, waarbij het proces over 3 paginaframes beschikt.

MK34 FIFO-vervanging, waarbij het proces over 4 paginaframes beschikt.

MK35 LRU-vervanging, waarbij het proces over 3 paginaframes beschikt.

MK36 LRU-vervanging, waarbij het proces over 4 paginaframes beschikt.

MK37 Working-set algoritme met working-set window = 4

MK38 Working-set algoritme met working-set window = 5

13. Filesystemen

Stel in een UNIX-systeem is in een directory een file-entry Y gecreëerd als *hard link* of als *soft (symbolic) link* naar een bestaande file X.
Geef aan voor de navolgende beweringen wat van toepassing is:

(A) alleen juist bij een hard link
(B) alleen juist bij een soft link
(C) juist voor zowel een hard link als een soft link
(D) onjuist voor beide typen link

MK39 X en Y mogen in dezelfde directory voorkomen

MK40 Door uitvoering van het commando "`rm X`" wordt de file X uit het filesysteem verwijderd.

MK41 Als X verwijderd wordt, dan wordt Y een "dangling link", die verwijst naar een niet bestaande file.

MK42 Als file X veranderd wordt, dan zal bij het lezen van file Y de oorspronkelijke informatie worden opgeleverd.

MK43 Achteraf is niet meer te onderscheiden of Y gecreëerd is als een link naar X óf X als een link naar Y.

MK44 X en Y kunnen in verschillende partities van een filesysteem voorkomen, als deze via een mount-point gekoppeld zijn.

14. Protectie

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

	F_1	F_2	D_1	D_2
D_1	read write X_1			X_2
D_2		read* write		

Voor de invulling van X_1 en X_2 zijn meerdere alternatieven gegeven, waarbij *empty* een lege invulling weergeeft.

(A) X_1 = empty, X_2 = empty
(B) X_1 = empty, X_2 = control
(C) X_1 = empty, X_2 = switch
(D) X_1 = read* , X_2 = switch
(E) X_1 = owner, X_2 = empty
(F) X_1 = owner, X_2 = control
(G) X_1 = owner, X_2 = switch

Geef voor elk van onderstaande operaties van een proces P in protectiedomein D_1 aan welk alternatief nodig is om deze operatie te mogen uitvoeren. Geef steeds het "zwakste" alternatief, d.w.z. met de minste rechten.

MK45 P verleent het write-recht op de file F1 aan het protectiedomein D2.

MK46 P verwijdert het write-recht op de file F2 uit het protectiedomein D2.

MK47 P verleent het read-recht op de file F2 aan het protectiedomein D1.

MK48 P verwerft tijdens zijn executie het write-recht op de file F2.

MK49 P kan een kopie maken van de file F1.

MK50 P kan op één moment de file F1 lezen en de file F2 schrijven.

15. UNIX

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

JN45 Om programma's te verwerken die een gebruiker als commando geeft, voert een shell i.h.a. een *fork*-system-call uit. Het nieuw gecreëerde proces bestaat daarbij uit een kopie van de adresruimte van [het shell-proces / het betreffende programma].

JN46 Het optreden van een *signal* heeft [wel / niet] effect op processen die naderhand op dit *signal* gaan wachten.

JN47 Een ouderproces kan via *pipes* met zijn zijn kindprocessen communiceren mits die *pipes* door het ouderproces [vóór / ná] de *fork*-operatie zijn gecreëerd.

JN48 De directorystructuur in UNIX-systemen is acyclisch dankzij het feit dat het gebruikers niet toegestaan is [*hard links* / *soft links*] naar een directory aan te maken.

JN49 In UNIX worden fileblokken via *inodes* geadresseerd. Afhankelijk van de grootte van de file wordt gebruik gemaakt van één of meer diskadressen in [dezelfde / verschillende] *inode(s)*.

JN50 Tijdens uitvoering van programma's waarvan het *setuid-bit* gezet is krijgt de [owner / gebruiker] van het programma tijdelijk de toegangsrechten van de [gebruiker / owner] van het programma.