Tentamen Operating Systems

Donderdag 26 april 2007, 14.00-17.00, examenhal

- Lees eerst een opgave volledig door, alvorens deze te maken.
- Schrijf netjes en zorgvuldig.
- Dit tentamen is 'Open Boek', d.w.z. dat het boek "*Operating Systems, Design and Implementation*" van Tanenbaum & Woodhull gebruikt mag worden als naslagwerk. Het is niet toegestaan ander materiaal, zoals college-aantekeningen en powerpoint-slides, te raadplegen.
- Het tentamen bestaat uit 6 opgaven. **Je hoeft echter slechts 5 opgaven te maken.** Kies zelf 5 opgaven!
- Iedere opgave is even veel punten waard.
- Je hebt 3 uur de tijd, gebruik deze nuttig. Ook als je snel klaar bent, gebruik dan de resterende tijd om jouw antwoorden nog eens te controleren.
- Succes!

Opgave 1: Scheduling
Het Unix besturingsysteem kent drie toestanden waarin een proces zich kan bevinden, namelijk de toestanden *Running*, *Ready* en *Blocked*.
(a) Geef van ieder van de onderstaande toestandsovergangen aan of deze mogelijk of onmogelijk is. Als de overgang mogelijk is, geef dan een voorbeeld van een situatie waarin deze overgang plaats zou kunnen vinden. Als de overgang niet mogelijk is, leg dan uit waarom.

1. Van *Running* naar *Ready*
2. Van *Running* naar *Blocked*
3. Van *Ready* naar *Blocked*
4. Van *Ready* naar *Running*

(b) Gegeven zijn 5 processen die zich aanmelden aan het besturingsysteem op verschillende tijdstippen. We gaan uit van tijdstippen in gehele seconden. Van ieder proces is het tijdstip van aanmelden (aankomst) en de duur van de executie bekend, en weergegeven in de volgende tabel.

proces	aankomst	executietijd
1	0	14
2	2	12
3	5	10
4	7	4
5	19	7

Bepaal de volgorde van executie van de processen en de gemiddelde wachttijd (waiting time) voor ieder van de volgende systemen.

- Systeem met non-pre-emptive *First Come First Served* (FCFS) scheduling
- Systeem met non-pre-emptive *Shortest Job First* (SJF) scheduling
- Systeem met pre-emptive *Shortest Remaining Time Next* (SRTN) scheduling met eem time quantum van 1 seconde.

Opgave 2: Deadlock preventie

In deze opgave beschouwen we een systeem met 4 resources (R_0, R_1, R_2 en R_3) en 3 processen (P_0, P_1, P_2). Van iedere resource is slechts een beperkte hoeveelheid beschikbaar, namelijk 6 eenheden van R_0, 4 eenheden van R_1, 7 eenheden van R_2 en 6 eenheden van R_3.

Stel nu dat ieder proces bij aanvang (start van het proces) aan het besturingsysteem aangeeft hoeveel eenheden het van iedere resource maximaal nodig zal hebben. Tevens is bekend dat een proces dat eenmaal alle eenheden toegekend heeft gekregen op den duur zal termineren en zijn verkregen resources zal vrijgeven (teruggave aan het besturingsysteem).

In een besturingsysteem waarin dit het geval is kan deadlock voorkomen worden met behulp van het zgn. *Banker's algorithm*. Veronderstel dat de toestand uit de volgende tabel is bereikt.

Resource	Totaal	Beschikbaar	Aanvraag			Toegekend		
R_i	aantal	aantal	P_0	P_1	P_2	P_0	P_1	P_2
0	6	3	3	1	1	1	1	1
1	4	1	3	2	1	2	0	1
2	7	1	2	3	5	2	3	1
3	6	2	2	4	0	1	3	0

De bovenstaande tabel dient als volgt gelezen te worden. Uit de eerste regel blijkt dat er 6 eenheden van resource R_0 in het systeem aanwezig zijn, waarvan op dit moment nog 3 eenheden beschikbaar. De reeds toegekende 3 eenheden zijn gelijk verdeeld over de processen P_0 tot en met P_2 (m.a.w. ieder 1 eenheid). Proces P_0 heeft maximaal 3 eenheden van resource R_0 aangevraagd, en kan in de toekomst dus nog ten hoogste 2 eenheden aanvragen. Processen P_1 en P_2 zullen geen eenheden van resource R_0 meer aanvragen.

(a) Een toestand heet *veilig* als het mogelijk is voor alle processen om te termineren (zonder deadlock). Laat zien dat de bovenstaande toestand veilig is.

(b) Schrijf een C of Java routine dat bepaalt of een gegeven toestand veilig is. Kies zelf een geschikte datarepresentatie voor de bovenstaande tabel.

(c) Leg uit hoe de routine uit onderdeel (b) gebruikt kan worden door het besturingsysteem om te bepalen of aanvragen voor resources door de processen gehonoreerd kunnen worden.

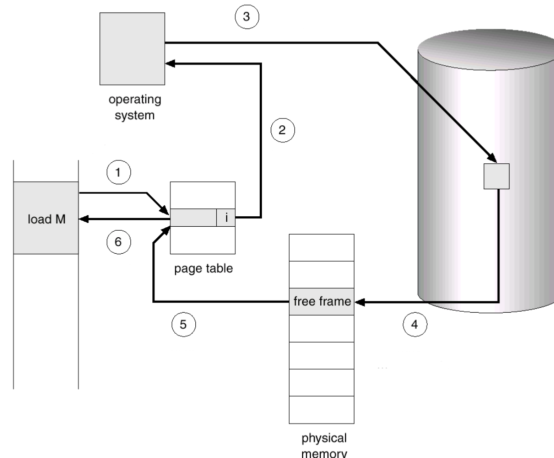
(d) Ga uit van de situatie in de bovenstaande tabel. Leg uit dat een aanvraag van proces P_2 voor 1 eenheid van resource R_2 veilig gehonoreerd kan worden. Geef de nieuwe toestand van het systeem na deze honorering.

(e) Ga uit van de situatie verkregen in onderdeel (d). Leg uit dat een aanvraag van proces P_1 voor 1 eenheid van resource R_1 niet gehonoreerd kan worden.

(f) Geef minstens drie redenen waarom het banker's algoritme in werkelijke besturingsysteem vrijwel nooit gebruikt wordt.

Opgave 3: Virtueel geheugen

(a) Leg uit hoe virtueel geheugen werkt aan de hand van de zes gemarkeerde punten uit de onderstaande figuur.



(b) Neem aan dat een processor een instructieset heeft waarbij iedere instructie 4 bytes lang is. Alignment van instructies is gebaseerd op even adressen, d.w.z. dat iedere instructie begint op een even adres. Iedere instructie heeft ten hoogste twee argumenten, waaronder de instructie MOV A,B. Deze instructie kopieert de inhoud van het woord (word size is 4 bytes) uit geheugenlocatie A in het woord op geheugenlocatie B.

Stel dat een besturingsysteem met virtueel geheugen voor deze processor aan een proces maximaal (slechts) 4 pageframes toekent. Leg uit dat het mogelijk is dat op een dergelijk systeem een proces in deadlock kan raken tgv. virtueel geheugenbeheer.

(c) Wat is het minimaal aantal pageframes dat het systeem aan een proces moet toekennen om het bovenstaande probleem te voorkomen? Licht het antwoord toe.

(d) Leg uit waarom ieder UNIX proces zijn eigen page table heeft. Geef aan wat de consequenties zouden zijn van een gemeenschappelijke page table op het niveau van de kernel.

(e) Een proces kan via systemcalls geheugen aanvragen en vrijgeven. Stel nu dat de enige actie die het besturingsysteem onderneemt het aanpassen van het valid-invalid-bit van de betreffende memory pages is. Leg uit waarom dit een potentieel veiligheidslek is. Hoe kan dit opgelost worden?

(f) Gegeven is de declaratie van de arrays `a` en `b`:

```
int a[1024][1024], b[1024][1204];
```

Beschouw de onderstaande twee lussen:

```
/* fragment 1: */
for (j = 0; j < 1024; j++)
  for (i = 0; i < 1024; i++)
    a[i][j] = b[i][j];


/* fragment 2: */
for (i = 0; i < 1024; i++)
  for (j = 0; j < 1024; j++)
    a[i][j] = b[i][j];
```

Beide fragmenten bewerkstelligen hetzelfde resultaat. Toch zal het ene fragment waarschijnlijk (veel) sneller zijn dat het andere fragment. Geef aan welk fragment het snelst zal zijn. Licht het antwoord toe.

Opgave 4: Page frame replacement
Ga in deze opgave uit van een virtueel memory systeem met slechts 4 page frames per proces. We beschouwen een proces dat memory memory frames benaderd volgens de volgende "reference sequence".

$$\text{ref}=[1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5].$$

Deze sequence geeft weer dat het proces gedurende executie eerst frame 1 gebruikt, vervolgens frame 2, daarna frame 3, etc.
(a) Leg uit waarom een optimaal page replacement algoritme voor dit proces slechts 6 pagefaults genereert. Geef na ieder van de bovenstaande 12 page-referenties aan welke 4 pages in het fysieke geheugen aanwezig zijn.
(b) Waarom zal een praktisch page replacement algoritme in de praktijk zelden tot het optimale (minimale) aantal pagefaults leiden?
(c) Ga uit van een FIFO page replacement algoritme. Geef voor ieder van de bovenstaande 12 page-referenties aan welke 4 pages in het fysieke geheugen aanwezig zijn. Wat is het aantal pagefaults?
(d) Het verschijnsel dat het aantal pagefaults kan toenemen terwijl het aantal beschikbare frames toeneemt heet *Belady's Anomaly.* Laat zien dat dit verschijnsel zich, gebruik makend van een FIFO replacement algoritme, voor de bovenstaande reference sequence ook kan voordoen.
(e) Ga uit van een LRU page replacement algoritme. Geef voor ieder van de bovenstaande 12 page-referenties aan welke 4 pages in het fysieke geheugen aanwezig zijn. Wat is het aantal pagefaults?

Opgave 5: Proces Synchronisatie
Semaforen zijn primitieven waarmee processen gesynchroniseerd kunnen worden. Er zijn drie operaties gedefinieerd op een semafoor, namelijk de operaties `P`, `V`, en `Init`.
(a) Leg uit wat deze operaties doen.
(b) Wat is een mutex? Leg uit hoe een semafoor eenvoudig gebruikt kan worden als een mutex.
(c) Het volgende klassieke synchronisatie probleem wordt toegeschreven aan Edgser Dijkstra.

We beschouwen een kapperszaak met één kapper, één kapperstoel en 3 stoelen voor wachtende klanten. Als er geen klanten zijn zit de kapper in zijn stoel en valt in slaap. Als een klant binnenkomt terwijl de kapper slaapt, dan wekt hij de kapper en neemt hij plaats in de kapperstoel. Als bij binnenkomst de kapper bezig is met een andere klant, dan neemt de klant plaats in één van de vrije stoelen en wacht op zijn beurt. Als er geen stoel beschikbaar is, dan vertrekt de klant weer.

Schrijf code (in Java, C, of pseudo-code) voor twee threads (processen met gemeenschappelijk geheugen). Het ene thread simuleert de kapper, terwijl het andere thread een klant simuleert (m.a.w. van het kapper-thread wordt slechts één instantiatie gemaakt, terwijl van het klant-thread meerdere instantiaties worden gemaakt). Je mag uitgaan van de beschikbaarheid van semaforen en de bijbehorende operaties. De oplossing dient vrij te zijn van deadlock en starvation.

Hint: Denk aan het producer-consumer algoritme. Een mogelijke oplossing maakt gebruik van 3 semaforen (waarvan één als mutex).

Opgave 6: Unix system programming

(a) De uitvoer van het onderstaande programma zal bij iedere executie waarschijnlijk verschillend zijn. Geef twee voorbeelden van mogelijke uitvoer.

(b) Leg uit waarom de uitvoer per executie kan verschillen.

(c) Leg kort (maar duidelijk) uit wat de code van de volgende regels bewerkstelligen:

- regel 13
- regels 18-21
- regel 23
- regel 26
- regel 30 (i.h.b. hoe kan `n<1` gelden?)
- regel 38

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <string.h>
 4  #include <unistd.h>
 5  #include <sys/wait.h>
 6  #include <fcntl.h>
 7
 8  int main (int argc, char *argv[]) {
 9    int fd[2], status;
10    int flags, n;
11    char msg[256];
12
13    if (pipe(fd) < 0) {
14      fprintf (stderr, "Could not make pipe\n");
15      return (EXIT_FAILURE);
16    }
17
18    flags = fcntl (fd[0], F_GETFL);
19    fcntl (fd[0], F_SETFL, flags | O_NONBLOCK);
20    flags = fcntl (fd[1], F_GETFL);
21    fcntl (fd[1], F_SETFL, flags | O_NONBLOCK);
22
23    if (fork() != 0) {
24      close(fd[0]);  /* fd[1] is used for reading */
25      sprintf (msg, "Hello world!\n");
26      write (fd[1], msg, 256); /* fd[1] is used for writing */
27    } else {
28      close(fd[1]);
29      do {
30        n = read (fd[0], msg, 256);
31        if (n<1) {
32          putchar ('#');
33          fflush(stdout);
34        }
35      } while (n<1);
36      printf (msg);
37    }
38    waitpid(-1, &status, 0);
39    return EXIT_SUCCESS;
40  }
```