Uitwerking Tentamen Operating Systems

Donderdag 26 april 2007, 14.00-17.00, examenhal

Opgave 1: Scheduling
(a) 1 punt

1. Van *Running* naar *Ready*: Mogelijk. De scheduler besluit dat een ander proces (in de ready toestand) de CPU toegewezen krijgt. Het running process gaat dan over in de ready toestand.
2. Van *Running* naar *Blocked*: Mogelijk. Het proces raakt geblokkeerd op een I/O-operatie.
3. Van *Ready* naar *Blocked*: Niet Mogelijk. Een process in de ready toestand kan niet een I/O-operatie doen en dus ook niet blokkeren.
4. Van *Ready* naar *Running*: Mogelijk. De scheduler wijst het proces de CPU toe.

(b) 1 punt Bij deze opgave is afronding van de gemiddelde wachttijd op gehele seconden goedgekeurd (hoewel strikt genomen niet juist). Met een quantum van 1 seconde zal er immers een gehele hoeveelheid aantal seconden door een proces gewacht worden.

- Systeem met non-pre-emptive *First Come First Served* (FCFS) scheduling: De volgorde is duidelijk 1, 2, 3, 4, 5 omdat het rijtje aankomsttijden stijgend is. De gemiddelde wachttijd is (0 + (14-2) + (14+12-5) + (14+12+10-7) + (14+12+10+4-19)/5= 16.6
- Systeem met non-pre-emptive *Shortest Job First* (SJF) scheduling De volgorde is nu 1, 4, 3, 5, 2. Merk op dat nadat processen 1 en 4 klaar zijn, er geldt t=18 < 19. Dus, proces 5 kan nog niet meedoen. Echter, als proces 3 klaar is, geldt t=28 > 19, en dus wordt dan proces 5 gekozen. De starttijden van de processen zijn: t=0 (proc. 1), t=14 (proc. 4), t=18 (proc 3), t=28 (proc. 5), t=35 (proc. 2). De gemiddelde wachttijd is dan: (0 + (35-2) + (18-5) + (14-7) + (28-19))/5 = 12.4
- Systeem met pre-emptive *Shortest Remaining Time Next* (SRTN) scheduling met eem time quantum van 1 seconde. De volgorde is nu iets complexer, namelijk 1, 4, 1, 3, 5, 3, 2. Het feit dat 1 en 3 beiden twee maal in deze lijst voorkomen is het gevolg van pre-emption. De gemiddelde wachttijd is (4+33+20+0+0)/5=11.4. De verantwoording hiervan blijkt uit de onderstaande tabel. Een entry van de vorm Rx/y in kolom k, voor een zekere t betekent dat na (!) tijdstip t het proces nog y seconden nodig heeft om te termineren, en in totaal x seconden in de ready state heeft gewacht. De R staat voor Running. Een wachtend process krijgt een entry Wx/y, met dezelfde betekenis voor x en y.

t	P1	P2	P3	P4	P5
1	R0/12				
4	R0/9	W3/12			
6	R0/7	W5/12	W2/10		
10	W4/7	W9/12	W6/10	R0/0	
17	R4/0	W16/12	W13/10		
18		W17/12	R13/9		
25		W24/12	W20/9		R0/0
34		W33/12	R20/0		
46		R33/0			

Opgave 2: Deadlock preventie
onderdelen: (a), (c)-(f) ieder 1/4 punt. onderdeel (b) 3/4 punt.
(a) De toestand is veilig, omdat de volgende sequentie van toekenningen leidt tot (deadlock-vrije) terminatie:

1. Ken [2,1,0,1] toe aan proces P_0. M.a.w. geef proces P_0 de beschikking over 2 eenheden R_0, 1 eenheid R_1, 0 eenheden R_2 en 1 eenheid R_3. Nu zal P_0 (uiteindelijk) termineren, en zijn resources weer vrijgeven. De nieuwe vector Available wordt dan Av=[4,3,3,3].
2. Ken [0,2,0,1] aan P_1 toe. De nieuwe Av is dan [5,3,6,6].
3. Ken tenslotte [0,0,4,0] toe aan P_2. Uiteraard is Av aan het einde gelijk aan het beschikbare aantal resources, m.a.w. [6, 4, 7, 6].

(b) Het onderstaande programma toont aan dat de toestand zoals gegeven in de tabel veilig is. Merk op dat slechts gevraagd werd om de routine `isSafe()` te schrijven.

```
#include <stdio.h>
#include <stdlib.h>

#define R 4 /* aantal resources */
#define P 3 /* aantal processen */

#define FALSE 0
#define TRUE  1

int available[R] = {3, 1, 1, 2};   /* beschikbare resources  */
int request[R][P] = {{3, 1, 1},    /* aangevraagde resources */
                     {3, 2, 1},
                     {2, 3, 5},
                     {2, 4, 0}};
int allocated[R][P] = {{1, 1, 1},  /* toegekende resources   */
                       {2, 0, 1},
                       {2, 3, 1},
                       {1, 3, 0}};

int isSafe(int available[R], int request[R][P], int allocated[R][P]) {
  int av[R];
  int finish[P];
  int r, p, np;

  /* bepaal processen die nog aanvragen open hebben staan */
  np = 0;
  for (p=0; p<P; p++) {
    for (r=0; r<R; r++)
      if (request[r][p] > allocated[r][p]) break;
    if (r<R) {
      np++;
      finish[p] = FALSE;
    } else {
      finish[p] = TRUE;
    }
  }

  /* kopieer available in av */
  for (r=0; r<R; r++)
```

```
    av[r] = available[r];

  while (np>0) {
    for (p=0; p<P; p++) {
      if (finish[p] == FALSE) {
        for (r=0; r<R; r++)
          if (request[r][p] - allocated[r][p] > av[r]) break;
        if (r==R) break;  /* er zijn voldoende resources voor p */
      }
    }
    if (p<P) {
      for (r=0; r<R; r++) av[r] += allocated[r][p];
      finish[p] = TRUE;
      np--;
    } else break;
  };
  return (np==0 ? TRUE : FALSE);
}


int main (int argc, char *argv[]) {
  printf ("%d\n", isSafe(available, request, allocated));
  return EXIT_SUCCESS;
}
```

(c) Stel een aanvraag komt binnen, in een zekere toestand. Het besturingsysteem bepaalt uit deze toestand de nieuwe toestand die bereikt zou worden als de aanvraag gehonoreerd zou zijn. Van deze nieuwe toestand word bepaald of deze veilig is, m.b.v. het algoritme uit onderdeel (b). Als dit het geval blijkt te zijn, dan wordt de toestand vervangen door de nieuwe toestand en de aanvraag gehonoreerd. Als de nieuwe toestand niet veilig is, dan wordt de oude toestand hersteld, en de aanvraag niet gehonoreerd.

(d) De nieuwe toestand, na honorering van een aanvraag van proces P_2 voor 1 eenheid R_2, wordt:

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	0	2	3	5	2	3	2
3	6	2	2	4	0	1	3	0

Omdat de oorspronkelijke toestand reeds veilig was, moet de nieuwe het ook zijn. Immers, P_0 en P_1 hebben reeds hun maximale hoeveelheid van deze resource gehonoreerd gekregen. M.a.w. er kan geen deadlock (meer) ontstaan op deze resource.

(e) De nieuwe toestand, na honorering van een aanvraag van proces P_1 voor 1 eenheid van R_1, zou worden:

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	0	3	2	1	2	1	1
2	7	0	2	3	5	2	3	2
3	6	2	2	4	0	1	3	0

Deze toestand is niet veilig. Immers, resource R_1 is nu uitgeput, terwijl P_1 nog 1 eenheid nodig heeft om zelf te termineren. M.a.w. proces P_1 kan niet vereder. Ook proces P_0 kan niet verder,

want die heeft ook nog 1 eenheid R_1 nodig. Tenslotte kan P_2 niet verder, omdat dit proces nog 3 eenheden R_2 nodig heeft, die niet beschikbaar zijn (die moeten vrijgegeven worden door P_0 en P_1). M.a.w. dit systeem zal in deadlock raken.

(f) Het banker's algoritme wordt in de praktijk niet gebruikt omdat

1. Het is een onrealistische eis dat ieder proces a priori aangeeft hoeveel het van een zekere resource nodig zal hebben.

2. In een wekelijk besturingsysteem varieert het aantal processen (en mogelijk zelfs resources) dynamisch.

3. De garantie dat processen op den duur (bij terminatie) resources vrijgeven is voldoende voor correctheid van het algoritme, maar is niet praktisch. Een proces dat enige uren (of dagen) rekent, alvorens zijn resources vrij te geven, zal leiden tot onacceptabele wachttijden.

Opgave 3: Virtueel geheugen
onderdeel (a) 3/4 punt, (b)-(f) ieder 1/4 punt.
(a) Toelichting bij de 6 punten uit de figuur.

1. Een proces refereert memory page M.

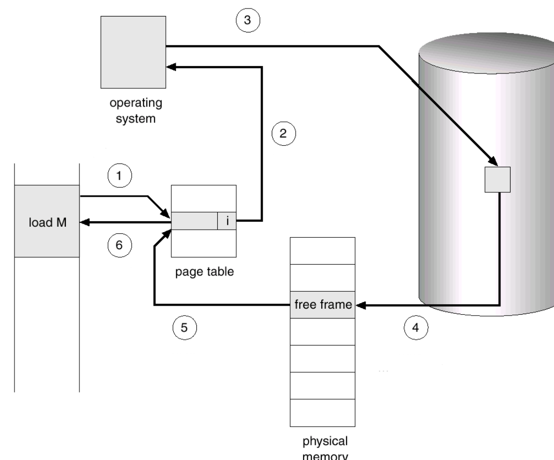
2. Uit de invalid/valid tabel blijkt dat page M niet in het geheugen aanwezig is. Er volgt een trap naar kernelniveau.

3. De kernel kiest een page die naar disk geschreven wordt, om zo ruimte te maken om page M van disk te laden.

4. Page M wordt van disk geladen in fysiek geheugen.

5. De page table wordt aangepast. Page M is nu valid.

6. Geef control terug aan het proces (herstart instructie).



(b)+(c) De instructie en de argumenten (ieder 4 bytes lang) zouden allen over de grens van verschillende pages kunnen liggen (eerste 2 bytes aan het einde van een page, en de volgende twee bytes aan het begin van de opvolgende page). M.a.w. de instructie levert pagefaults voor de instructie zelf, en voor de argumenten. Als er dus 4 pages beschikbaar zijn, dan raken we op deze instructie in een oneindege lus (oneindig veel page faults). Met (minimaal) 6 pages, is dit probleem opgelost.
(d) Ieder UNIX proces heeft zijn eigen page table heeft. Als de table gemeenschappelijk zou zijn, dan zouden processen niet meer de volledige maximale grootte van het virtuele geheugenbereik

kunnen addresseren, omdat andere processen reeds delen van die geheugenruimte in beslag hebben genomen. Er ontstaat competitie op geheugengebruik tussen de processen. Het is dan mogelijk dat op een zeker tijdstip een programma gebruikt kan worden, terwijl op een ander tijdstip dit niet mogelijk blijkt te zijn (wegens gebruik van een deel van de adresruimtege door andere processen).
(e) Een proces A vult een memory page M met gevoelige data (bijvoorbeeld tekst van een tekstverwerker). Als een tweede proces B (van een andere gebruiker) vervolgens geheugen aanvraagt, en het besturingsysteem besluit om de pagina M naar disk te schreven om de vrijgekomen page vervolgens te gebruiken voor proces B (en alleen de valid/invalid bit wordt gemodificeerd) dan zal proces B de data van proces A kunnen lezen. Een oplossing is om de page eerst te vullen met bijvoorbeeld nullen.
(f) Hoewel de arrays `a` en `b` twee-dimensionaal zijn, zal de fysieke geheugen layout een lineair array zijn, namelijk eerst rij 0, dan rij1, enz. Stel nu dat bijvoorbeeld één rij precies in een memory page past. Dan levert het tweede fragment 2*1024=2048 pagefaults op, terwijl het andere fragment 2*1024*1024=2097152 pagefaults op levert. Het tweede fragment zal daarom veel sneller zijn dan het eerste fragment.

Opgave 4: Page frame replacement onderdelen (a)+(c)+(e) ieder 1/2 punt, (b)+(d) ieder 1/4 punt.

(a) Een optimaal page replacement algoritme 'kijkt in de toekomst'. Als een page gekozen moet worden ter vervanging, dan wordt die page gekozen die het langst niet nodig zal zijn. Volgens deze methode komen we tot het onderstaande schema. Een bold-face getal geeft een pagefault weer. Uit de tabel blijkt dat er 6 pagefaults optreden.

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

(b) Het bovenstaande algoritme is optimaal. Echter het maakt gebruik van niet beschikbare informatie (je kunt nu eenmaal niet in de toekomst kijken). M.a.w. ieder praktisch algoritme zal waarschijnlijk slechter scoren omdat het geen gebruik kan maken van deze informatie.
(c) Een FIFO page replacement algoritme met een page table met 4 entries leidt tot de onderstaande tabel. We tellen 10 pagefaults.

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

(d) Een FIFO page replacement algoritme met een page table met 3 entries leidt tot de onderstaande tabel. We tellen 9 pagefaults. Terwijl de pagetable kleiner is, blijkt het aantal pagefaults ook kleiner te zijn.

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

(e) Een LRU page replacement algoritme met een page table met 4 entries leidt tot de onderstaande tabel. We tellen 8 pagefaults.

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

Opgave 5: Proces Synchronisatie
onderdelen (a)+(b) ieder 1/2 punt, (c) 1 punt.
(a) Een semafoor kan beschouwd worden als een integer variabele waarop operaties atomair zijn. De operatie `P(s)` zal de waarde van de semafoor `s` proberen te verlagen (maar blokkeert als `s=0`). De operatie `V(s)` verhoogt de waarde van de semafoor. Met `Init(s, val)` kan de semafoor `s` met de waarde `val` geïnitialiseerd worden. De operaties kunnen als volgt gekarakteriseerd worden:

```
P(s)=|[ <await s>0>; s-- ]|
V(s)=|[ s++; ]|
Init(s,val)=|[ s=val; ]|
```

(b) Een mutex is een atomaire variabele die slechts twee waarden kan aannemen, namelijk 0 (unlocked) of 1 (locked). De operatie `lock(m)` probeert de waarde van de mutex op 1 te zetten. Als de waarde reeds 1 was, dan blokkeert deze operatie. De operatie `unlock(m)` zet de waarde van de mutex op 0. Deze operatie mag alleen uitgevoerd worden door dat proces dat oorspronkelijk ook de lock heeft gezet (m.a.w. de mutex bezit).

Locks zijn eenvoudig te implementeren met een semafoor die de waarden 0 en 1 mag aannemen. De operatie `lock(m)` is dan te implementeren als `P(m)`, terwijl de afspraak is dat `V(m)` alleen uitgevoerd mag worden door het proces dat de lock bezit.
(c) De onderstaande code is een simulatie van het 'sleeping barber's problem'.

```
Semafoor semKlant;  /* intieel 0 */
Semafoor semKapper; /* intieel 0 */
Semafoor mutStoel;  /* intieel 0 (unlocked) */

int vrijeStoelen;  /* initieel 3 */

void KapperThread () {
  while(1) { /* oneindige lus */
    P(semKlant);   /* nieuwe klant of slapen */
    P(mutStoel);   /* lock voor modificeren aantal stoelen */
    vrijeStoelen++;
    V(semKapper);  /* kapper is beschikbaar */
    V(mutStoel); /* geef toegang tot vrijeStoelen vrij */
    knipHaar();
}

void klantThread () {
  P(mutStoel); /* atomair toegang tot vrijeStoelen */
  if (vrijeStoelen>0) {
    /* er is een stoel vrij */
    vrijeStoelen--;
    V(semKlant);  /* signaleer aan de kapper dat er een klant is */
    V(mutStoel);  /* geef toegang tot vrijeStoelen vrij */
    P(semKapper); /* wacht als de kapper bezig is */
  } else {
    /* geen vrije stoelen */
    V(mutStoel); /* geef toegang tot vrijeStoelen vrij */
    return;      /* vertrek */
  }
}
```

Opgave 6: Unix system programming
onderdelen (a)+(b) ieder 1/4 punt, (c) 1.5 punt.

(a)+(b) De uitvoer bestaat uit 0 of meer '#' gevolgd door de tekst `Hello world!`. Twee mogelijke uitvoeren zijn dus "`Hello world!` en `#####Hello world!`".

Het programma bestaat uit een parent en een kind proces die communiceren via een non-blocking pipe. Doordat deze pipe non-blocking is, is het resultaat van de read-operatie op regel 30 non-deterministisch (en dus scheduling afhankelijk).

(c)

- regel 13: Een uni-directioneel communicatiekanaal (pipe) wordt gemaakt. Dit levert een klein array van twee file-descriptoren. Descriptor `fd[0]` kan gebruikt worden voor lees-operaties, terwijl `fd[1]` gebruikt kan worden voor schrijf-operaties.
- regels 18-21: De control-flag van de pipe die aangeeft dat read/write-operaties op de pipe non-blocking moeten zijn wordt gezet.
- regel 23: Het kind proces wordt gemaakt d.m.v. de systemcall `fork`. Het kind proces is een exact duplicaat van het parent-proces. De syscall `fork` retourneert de waarde 0 aan het kind, terwijl de parent het proces-id (pid) van het child-proces krijgt.
- regel 26: De boodschap wordt in de pipe geschreven door de parent.
- regel 30: Het kind-proces leest (non-blocking!) van de pipe. De returnwaarde geeft het aantal gelezen bytes aan. Als deze 0 is, dan betekent dit dat de parent nog niets heeft geschreven.
- regel 38: Het proces wacht op terminatie van een kind. M.a.w de parent wacht op de terminatie van het kind, terwijl het kind zelf geen kinderen heeft en dus niet wacht.