Tentamen Operating Systems

Woensdag 23 januari 2008, 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 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
Gegeven zijn 5 processen die zich aanmelden aan het besturingsysteem op verschillende tijdstippen. Alle processen zijn volledig CPU-bound. 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
A	0	9
B	2	6
C	3	3
D	4	4
E	9	8

(a) Bepaal de volgorde van executie van de processen en de gemiddelde *turnaround 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 *Round-Robin Scheduling* scheduling met eem time quantum van 3 seconden.
- Systeem met pre-emptive *Shortest Remaining Time Next* (SRTN) scheduling met een time quantum van 2 seconden.

(b) We beshouwen nu priority scheduling met 3 prioriteitsniveaus. De bovenstaande processen hebben de volgende prioriteiten:

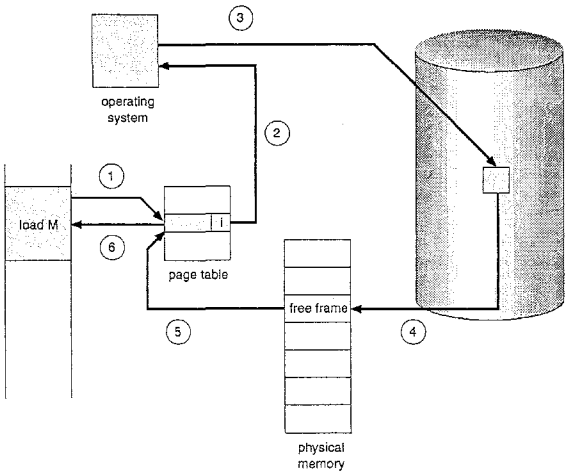
- hoogste prioriteit: processen B en D
- midden prioriteit: processen E en C
- laagste prioriteit: proces A

Bepaal de volgorde van executie van de processen en de gemiddelde *turnaround time* voor een systeem met pre-emptive priority scheduling met een time quantum van 2 seconden. Ga er vanuit dat iedere klasse round robin wordt gescheduled.

(c) Een user-level thread library maakt het mogelijk om binnen een UNIX-proces meerdere threads te maken zonder dat de kernel dit kan zien. M.a.w. voor de scheduler bestaat er slechts één proces terwijl binnen dit proces meerdere threads kunnen bestaan. Leg uit hoe dit problemen kan leveren met een standaard round robin scheduler voor een proces dat bestaat uit een thread dat CPU-bound is en een thread dat I/O-bound is.

Opgave 2: Virtueel geheugen

(a) Leg uit hoe virtueel geheugen werkt aan de hand van de 6 gemarkeerde punten uit de volgende figuur.



(b) Waarom is de grootte van een pageframe altijd een macht van twee?

(c) Bij ieder virtueel geheugen systeem is het mogelijk om van bepaalde memory pages aan te geven dat zij niet naar disk geswapped mogen worden. Leg uit waarom dit is en geef een voorbeeld.

(d) Beschouw een computer met 32 bits geheugenaddressering waarop een besturingsysteem draait met virtueel geheugen. De memory pages van dit systeem zijn 4KB groot. Geef commentaar op de volgende pagetable groottes:

- een pagetable van $2^{19} = 524288$ entries.
- een pagetable van $2^{20} = 1048576$ entries.
- een pagetable van $2^{21} = 2097152$ entries.

(e) Een gebruikelijke richtlijn voor het kiezen van de grootte van een swappartitie op een disk is twee maal de omvang van het fysiek aanwezige geheugen. Heeft het installeren van 6 GB swapspace op een 32 bits machine met 3 GB geheugen zin? Licht je antwoord toe.

(f) Beschouw een systeem dat in de situatie is geraakt waarin voortdurend pagefaults plaats vinden (een z.g.n. thrashing state). Dit kan opgelost worden door de scheduler één of meerdere processen te laten kiezen die voorlopig niet de beschikking over de CPU zullen krijgen. Geef drie voorbeelden van mogelijke criteria waarop een besturingsysteem zou kunnen besluiten welke processen dit zijn. Geef bij ieder voorbeeld aan of er extra informatie bijgehouden dient te worden door het OS.

Opgave 3: Page frame replacement

Ga in deze opgave uit van een virtueel memory systeem met slechts 3 page frames per proces. We beschouwen een proces dat memory pages benadert volgens de volgende "reference sequence".

$$\text{ref}=[1,\ 2,\ 3,\ \underline{4},\ 1,\ 5,\ \underline{6},\ 2,\ 5,\ 6,\ \underline{1},\ 2,\ 3,\ \underline{4},\ 5].$$

Deze sequence geeft weer dat het proces gedurende executie eerst page 1 gebruikt, vervolgens page 2, daarna page 3, etc. Een aantal pages is onderstreept. Dit speelt pas een rol in onderdeel (d).

(a) Bepaal het minimaal aantal pagefaults, m.a.w. het aantal pagefaults dat een optimaal page replacement algoritme voor dit proces genereert. Geef na ieder van de bovenstaande 15 page-referenties aan welke pages in het fysieke geheugen aanwezig zijn.

(b) Ga uit van een FIFO page replacement algoritme. Geef voor ieder van de bovenstaande 15 page-referenties aan welke 3 pages in het fysieke geheugen aanwezig zijn. Wat is het aantal pagefaults?

(c) Is het mogelijk om met het "Second chance Page Replacement" algoritme meer pagefaults te krijgen dan met het FIFO-algoritme? Licht jouw antwoord toe.

(d) Veronderstel nu dat het "aging algoritme" wordt gebruikt voor page replacement (zie fig. 4-17 in het boek). De klok-tikken vinden plaats juist voordat de onderstreepte pages worden gerefereerd. We gaan uit van age-counters van 4 bits. Hoeveel pagefaults levert dit page replacement algoritme voor de bovenstaande reference sequence?

Opgave 4: Proces Synchronisatie

Semaforen zijn primitieven waarmee processen gesynchroniseerd kunnen worden. Er zijn twee operaties gedefinieerd op een semafoor, namelijk de operaties `down` en `up`.

(a) Leg uit wat deze operaties doen.

(b) Implementeer zelf semaforen (in Java, C++ of C) waarbij je gebruik mag maken van een routine `TSL(rx,lock)`. Deze routine implementeert de z.g.n. *Test and Set Lock* operatie (zie blz. 75 in het boek). Je mag gebruik maken van busy-waiting.

(c) We beschouwen verschillende implementaties van een eindige FIFO-buffer met één producer en één consumer. De synchronisatie geschiedt met semaforen. De semaforen en een functie `producer()` zijn reeds gegeven (zie blz. 80 van het boek):

```
#define N 100 /* number of slots in the buffer */
semaphore mutex = 1, empty = N, full = 0;

void producer() {
  int item;
  while (1) {
    item = produce_item();    /* generate item                 */
    down(&empty);             /* decrement empty count         */
    down(&mutex);             /* enter critcal region          */
    insert_item(item);        /* put item in buffer            */
    up(&mutex);               /* leave critical region         */
    up(&full);                /* increment count of ufll slots */
  }
}
```

We beschouwen nu de onderstaande 4 implementaties van de routine `consumer()`. Beoordeel de verschillende implementaties. Geef van iedere implementatie aan of deze correct is. Leg uit waarom wel/niet.

```
void consumer1() {
  while (1) {             /* infinite loop                 */
    down(&mutex);         /* enter critical region         */
    down(&full);          /* decrement full count          */
    item=remove_item();   /* take item from buffer         */
    up(&mutex);           /* leave critical region         */
    up(&empty);           /* increment count of empty slots */
    consume_item(item);   /* do something with the item    */
  }
}

void consumer2() {
  while (1) {             /* infinite loop                 */
    down(&mutex);         /* enter critical region         */
    down(&full);          /* decrement full count          */
    item=remove_item();   /* take item from buffer         */
    up(&empty);           /* increment count of empty slots */
    up(&mutex);           /* leave critical region         */
    consume_item(item);   /* do something with the item    */
  }
}

void consumer3() {
  while (1) {             /* infinite loop                 */
    down(&full);          /* decrement full count          */
    down(&mutex);         /* enter critical region         */
    item=remove_item();   /* take item from buffer         */
    up(&mutex);           /* leave critical region         */
    up(&empty);           /* increment count of empty slots */
    consume_item(item);   /* do something with the item    */
  }
}
```

```
void consumer4() {
  while (1) {              /* infinite loop                  */
    down(&full);           /* decrement full count           */
    down(&mutex);          /* enter critical region          */
    item=remove_item();    /* take item from buffer          */
    up(&empty);            /* increment count of empty slots */
    up(&mutex);            /* leave critical region          */
    consume_item(item);    /* do something with the item     */
  }
}
```

Opgave 5: Unix system programming
In deze opgave houden we ons bezig met een eindige variant van het klassieke "dining philosophers problem". Het probleem laat zich als volgt beschrijven. Aan een ronde tafel zitten 5 filosofen voor een bord dampende spaghetti. Elke filosoof herhaalt 3 keer: denken, eten. Tussen twee borden ligt er één vork. Omdat filosofen netjes en beleefd zijn eten ze met twee vorken. Dit houdt dan wel in dat twee naburige filosofen afwisselend eenzelfde vork moeten gebruiken. En dit kan problemen geven. Het probleem bestaat er uit een protocol te maken zodanig dat geen van de filosofen een hongerdood zal sterven.

Het onderstaande programma is een poging de filosofen te simuleren met 5 UNIX-processen.

(a) Bestudeer de code nauwkeurig. Leg kort de structuur van het programma uit. Maak eventueel een eenvoudige schets van de processen.
(b) Bij iedere executie van het programma zal de uitvoer waarschijnlijk verschillend zijn. Geef twee voorbeelden van mogelijke uitvoer.
(c) Leg uit waarom de uitvoer per executie kan verschillen.
(d) Leg kort (maar duidelijk) uit wat de code van de volgende regels bewerkstelligen:

- regels 53-54
- regels 55-56
- regel 57
- regel 8
- regel 36 (i.h.b. hoe kan `<=0` gelden?)
- regel 60

(e) Termineert het programma altijd? Zo ja, waarom. Zo nee, geef een scenario waarin het programma niet termineert.

```
1 int takefork(int send, int recv, int f) {
2   write(send, &f, sizeof(int));
3   read(recv, &f, sizeof(int));
4   return f;
5 }
6
7 void putfork(int send, int f) {
8   write(send, &f, sizeof(int));
9 }
10
11 void philosopher(int whoami, int send, int recv) {
12   int gotboth, msg, cnt=0, left = whoami, right = (whoami+1)%5;
13   while (cnt<3) {
14     printf ("%d: Thinking\n", whoami);
15     gotboth = 0;
16     while (gotboth == 0) {
17       while (takefork(send, recv, left) != whoami);
18       if (takefork(send, recv, right) == whoami)
19         gotboth = 1;
20       else putfork(send, left);
21     }
22     printf ("%d: Eating\n", whoami);
23     putfork(send, left);
24     putfork(send, right);
25     cnt++;
26   }
27   msg = -1;
28   write(send, &msg, sizeof(int));
29   exit(EXIT_SUCCESS);
30 }
31
32 void manageForks(int toFork[5][2], int fromFork[5][2]) {
33   int p, msg, done=0, owner[5]={-1,-1,-1,-1,-1};
34   while (done<5) {
35     for (p=0; p<5; ++p) {
36       if (read(toFork[p][0], &msg, sizeof(int)) > 0) {
37         if (msg == -1) done++;
38         else {
39           if (owner[msg] == p) owner[msg] = -1;
40           else {
41             if (owner[msg] == -1) owner[msg] = p;
42             write(fromFork[p][1], &owner[msg], sizeof(int));
43           }
44         }
45       }
46     }
47   }
48 }
49
50 int main(int argc, char *argv[]) {
51   int p, flags, status, toFork[5][2], fromFork[5][2];
52   for (p=0; p<5; p++) {
53     pipe(toFork[p]);
54     pipe(fromFork[p]);
55     flags = fcntl (toFork[p][0], F_GETFL);
56     fcntl (toFork[p][0], F_SETFL, flags | O_NONBLOCK);
57     if (fork() == 0) philosopher(p, toFork[p][1], fromFork[p][0]);
58   }
59   manageForks(toFork, fromFork);
60   for (p=0; p<5; p++) waitpid(-1, &status, 0);
61   return EXIT_SUCCESS;
62 }
```