

Onderzoeksverslag

– Semaforen –



Student:	A.W. Janisse
Studentnummer:	2213829
Instelling:	Fontys hogescholen te Eindhoven
Opleiding:	ICT & Technology
Docent:	Dhr. Erik Dortmans

Opdracht:	4, Semaforen
Datum:	16 maart 2015

Inleiding

In deze opdracht wordt onderzocht hoe de acties van processen (of threads) gesynchroniseerd kunnen worden en mutual exclusion gerealiseerd kan worden door middel van semaforen.

Testprogramma

Om de opdrachten uit te voeren is er een C programma gebruikt. Dit programma is als source aangeleverd door school en door mij zelf gecompileerd conform de opdracht met: `gcc sem.c -o sem -lrt`. Dit blijkt dus niet goed te zijn. Afbeelding 1 laat het resultaat zien.

```
student@student-fontys:~/bart/programming/school/opdracht.2/IPC32_ipc$ gcc sem.c -o sem -lrt
sem.c: In function 'main':
sem.c:82:17: warning: format '%x' expects argument of type 'unsigned int', but argument 2 has type 'union sem_t *' [-Wformat=]
    printf ("sem_open() returned %#x\n", semdes);
    ^
sem.c:94:17: warning: format '%o' expects a matching 'unsigned int' argument [-Wformat=]
    printf ("Calling sem_open('%s', 0, %#o)\n", sem_name);
    ^
sem.c:101:17: warning: format '%x' expects argument of type 'unsigned int', but argument 2 has type 'union sem_t *' [-Wformat=]
    printf ("sem_open() returned %#x\n", semdes);
    ^
/tmp/cctDQzME.o: In function 'main':
sem.c:(.text+0x144): undefined reference to `sem_open'
sem.c:(.text+0x1e9): undefined reference to `sem_open'
sem.c:(.text+0x251): undefined reference to `sem_close'
sem.c:(.text+0x2c1): undefined reference to `sem_post'
sem.c:(.text+0x329): undefined reference to `sem_trywait'
sem.c:(.text+0x391): undefined reference to `sem_wait'
sem.c:(.text+0x401): undefined reference to `sem_getvalue'
sem.c:(.text+0x48e): undefined reference to `sem_unlink'
collect2: error: ld returned 1 exit status
student@student-fontys:~/bart/programming/school/opdracht.2/IPC32_ipc$
```

Afbeelding 1

In deze afbeelding is duidelijk te zien dat in ieder geval geen referentie gevonden is naar `SEM_XXX`. Het raadplegen van de man pages¹ heeft geleerd dat het programma gelinkt moet worden tegen `pthread` library. Afbeelding 2 laat zien wat de opdracht `gcc sem.c -o sem -lpthread -lrt` oplevert. Nog niet goed dus.

```
student@student-fontys:~/bart/programming/school/opdracht.2/IPC32_ipc$ gcc sem.c -o sem -pthread -lrt
sem.c: In function 'main':
sem.c:82:17: warning: format '%x' expects argument of type 'unsigned int', but argument 2 has type 'union sem_t *' [-Wformat=]
    printf ("sem_open() returned %#x\n", semdes);
    ^
sem.c:94:17: warning: format '%o' expects a matching 'unsigned int' argument [-Wformat=]
    printf ("Calling sem_open('%s', 0, %#o)\n", sem_name);
    ^
sem.c:101:17: warning: format '%x' expects argument of type 'unsigned int', but argument 2 has type 'union sem_t *' [-Wformat=]
    printf ("sem_open() returned %#x\n", semdes);
    ^
student@student-fontys:~/bart/programming/school/opdracht.2/IPC32_ipc$
```

Afbeelding 2

¹ Programs using the POSIX semaphores API must be compiled with `cc -pthread` to link against the real-time library, `librt`. [man]

Afbeelding 3 laat de stukjes van de code zien waarin het mis gaat (Regel 82, 94 en 101).

```
75     printf ("Calling sem_open('%s', 0_CREAT | 0_EXCL)\n", sem_name);
76     semdes = sem_open (sem_name, 0_CREAT | 0_EXCL, 0600, 1);
77
78     if (semdes == SEM_FAILED)
79     {
80         perror ("ERROR: sem_open() failed");
81     }
82     printf ("sem_open() returned %#x\n", semdes);
83     break;
84 case 'o':
85     if (semdes != SEM_FAILED)
86     {
87         printf ("ERROR: another semaphore already opened\n");
88         break;
89     }
90     printf ("Enter name: ");
91     fgets (sem_name, sizeof (sem_name), stdin);
92     remove_nl (sem_name);
93
94     printf ("Calling sem_open('%s', 0, 0)\n", sem_name);
95     semdes = sem_open (sem_name, 0);
96
97     if (semdes == SEM_FAILED)
98     {
99         perror ("ERROR: sem_open() failed");
100     }
101     printf ("sem_open() returned %#x\n", semdes);
102     break;
```

Afbeelding 3

Zoals het resultaat van het compileren al liet zien (afbeelding 2) zit het probleem in de verwachte parameter type zoals die opgegeven is bij *printf(...)* functie.

Om het probleem op te lossen zijn de genoemde regels respectievelijk als volgt aangepast:

```
81     printf ("sem_open() returned %#x\n", (unsigned int)semdes);
82     break;
83
84     printf ("Calling sem_open('%s', 0)\n", sem_name);
85
101     printf ("sem_open() returned %#x\n", (unsigned int)semdes);
```

Na deze aanpassingen compileert het programma goed en is zoals afbeelding 4 laat zien ook executeerbaar.

```
student@student-fontys:~/bart/programming/school/opdracht.2/IPC32_ipc$ ./sem
Menu
=====
[n] create new semaphore
[o] open existing semaphore
[p] post
[t] trywait
[w] wait
[g] getvalue
[c] close
[u] unlink
[q] quit
Enter choice: █
```

Afbeelding 4

Opdrachten

Als eerste opdracht start ik het programma meerdere keren op in verschillende shells en speel een scenario af (bijvoorbeeld: ene proces creëert een semaphore, de ander opent het; ene proces wacht op een semaphore, de ander laat passeren) . Hierbij beantwoord ik de volgende vragen:

- Beschrijf een scenario dat je uitgevoerd hebt, en wat je dan waarneemt.
- Waar kan je de semaphore in het filesysteem terugvinden?
- Wat zie je als je een hexdump (zie od) van die file maakt? Check dit als je enkele keren 'post' hebt gedaan, en daarna nogmaals als je enkele keren 'wait' hebt gedaan (zodat alle processen geblokkeerd zijn).

Om antwoord te geven op de bovenstaande vragen, zal ik eerst onderzoeken wat de verschillende menu opties precies doen. Omdat de meeste van deze opties wat mij betreft evident zijn, is het onderstaande overzicht gemaakt:

- [w] **wait:**
Roept de functie `sem_wait()`² aan. Deze functie zal normaliter vlak voor de *critical section* worden aangeroepen. Door het aanroepen van deze functie zal mutual exclusion worden gegarandeerd. Intern zal deze functie proberen een 'lock' te krijgen door het verlagen van een teller. Als dit niet lukt dan zal er gewacht worden tot dit wel kan.
- [t] **try wait:**
Roept de functie `sem_trywait()`³ aan. De functie probeert het zelfde als de `wait()` functie en zal dus proberen om een 'lock' te krijgen. Het verschil zit hem er in dat als het verkrijgen van de 'lock' niet slaagt, dat de functie terug keert met een fout code in plaats van te wachten.
- [p] **post:**
Roept de functie `sem_post()`⁴ aan. Deze functie zal normaliter na het verlaten van een *critical section* worden aangeroepen. Eventueel een wachtend ander proces zal hierdoor de *critical section* mogen uitvoeren.
- [g] **getvalue:**
Roept de functie `sem_getvalue()`⁵ aan. Deze functie kan worden gebruikt om de waarde van de semafoor op te vragen waarmee uiteindelijk het aantal geblokkeerde processen of threads wordt aangeduid.

-
- 2 **sem_wait()** decrements (locks) the semaphore pointed to by *sem*. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call. [man].
- 3 **sem_trywait()** is the same as **sem_wait()**, except that if the decrement cannot be immediately performed, then call returns an error (*errno* set to **EAGAIN**) instead of blocking. [man].
- 4 **sem_post()** increments (unlocks) the semaphore pointed to by *sem*. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a [sem_wait\(3\)](#) call will be woken up and proceed to lock the semaphore [man].
- 5 **sem_getvalue()** places the current value of the semaphore pointed to *sem* into the integer pointed to by *sval*. either 0 is returned; or a negative number whose absolute value is the count of the number of processes and threads currently blocked in [sem_wait\(3\)](#). [man]

Antwoord op de vragen (2.1 POSIX Semaphores)

Met betrekking tot eerste vraag voer ik het volgende scenario uit:

P0:

Kies optie n (create new)

Kies optie p (post)

Kies optie c (close)

Kies optie u (unlink)

P1:

Kies optie o (open semafoor)

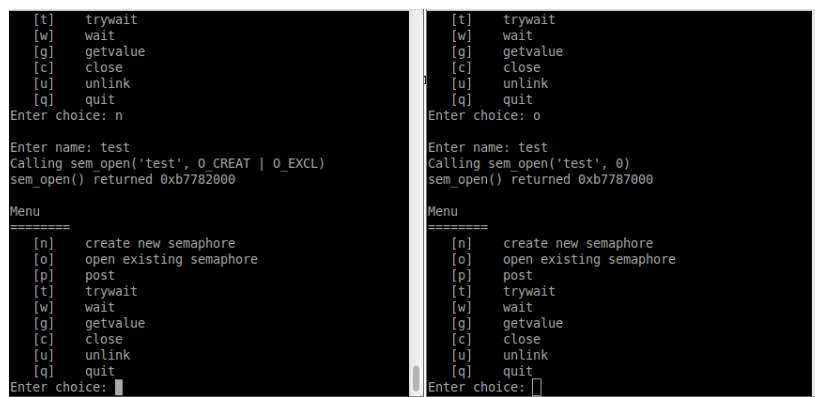
Kies optie w (wait)

Kies optie w (wait)

Kies optie c (close)

Kies optie u (unlink)

Afbeelding 5 laat zien wat het resultaat is van het aanmaken en het openen van de semafoor. Het aanmaken is geslaagd en het openen is geslaagd. Beide processen laten het zelfde adres (0xb7782000) van de nieuwe semafoor zien.



```
[t] trywait
[w] wait
[g] getvalue
[c] close
[u] unlink
[q] quit
Enter choice: n

Enter name: test
Calling sem_open('test', 0_CREAT | 0_EXCL)
sem_open() returned 0xb7782000

Menu
=====
[n] create new semaphore
[o] open existing semaphore
[p] post
[t] trywait
[w] wait
[g] getvalue
[c] close
[u] unlink
[q] quit
Enter choice: [ ]

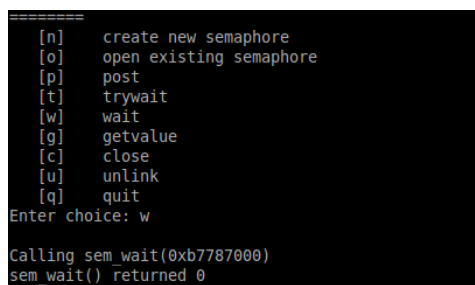
[t] trywait
[w] wait
[g] getvalue
[c] close
[u] unlink
[q] quit
Enter choice: o

Enter name: test
Calling sem_open('test', 0)
sem_open() returned 0xb7782000

Menu
=====
[n] create new semaphore
[o] open existing semaphore
[p] post
[t] trywait
[w] wait
[g] getvalue
[c] close
[u] unlink
[q] quit
Enter choice: [ ]
```

Afbeelding 5

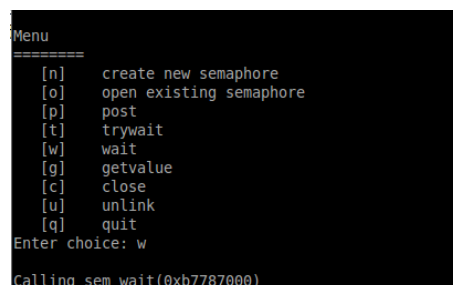
Vervolgens is voor P1 de w optie gekozen. Zoals verwacht zal deze direct terug keren (returnd 0) omdat er nog geen 'lock' is. Wel zal de waarde van de semafoor verlaagd zijn. Zie afbeelding 7. Direct daarna wordt nogmaals de op w gekozen. Omdat bij de eerste keer aanroepen van `sem_wait()` de waarde van de semafoor verlaagd is, zal het proces bij de tweede aanroep dus wachten totdat de 'lock' verkregen wordt. Afbeelding 6 laat hiervan het resultaat zien.



```
=====
[n] create new semaphore
[o] open existing semaphore
[p] post
[t] trywait
[w] wait
[g] getvalue
[c] close
[u] unlink
[q] quit
Enter choice: w

Calling sem_wait(0xb7782000)
sem_wait() returned 0
```

Afbeelding 7

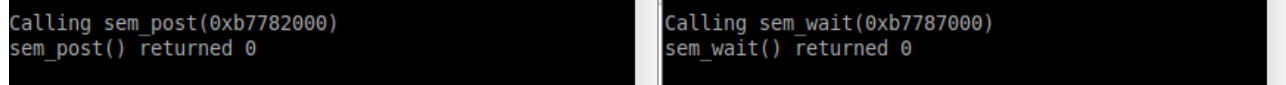


```
Menu
=====
[n] create new semaphore
[o] open existing semaphore
[p] post
[t] trywait
[w] wait
[g] getvalue
[c] close
[u] unlink
[q] quit
Enter choice: w

Calling sem_wait(0xb7782000)
```

Afbeelding 6

Na het kiezen van de optie p bij proces 1 zal de 'lock' worden opgeheven. Het resultaat hiervan is dat P1 terug keert van de aanroep van `sem_wait()`. Afbeelding 8 laat hiervan het resultaat zien.



The image shows two side-by-side terminal windows. The left window displays the text 'Calling sem_post(0xb7782000)' followed by 'sem_post() returned 0'. The right window displays 'Calling sem_wait(0xb7787000)' followed by 'sem_wait() returned 0'. Both windows have a black background with white text.

```
Calling sem_post(0xb7782000)
sem_post() returned 0
```

```
Calling sem_wait(0xb7787000)
sem_wait() returned 0
```

Afbeelding 8

Vervolgens zijn voor beide processen de semafoor succesvol gesloten en ge-unlinkt. De conclusie voor dit scenario is als volgt: Zodra er eenmaal een 'lock' is en een proces roept een `sem_wait()` functie aan dan zal dit proces wachten totdat deze een 'lock' kan aanmaken. Deze 'lock' kan pas aangemaakt worden als er een `sem_post()` wordt aangeroepen.

Om uit te zoeken hoe de functie `sem_trywait()` zich precies gedraagt ga ik het volgende scenario uitvoeren:

P0:

Kies optie n (create new)

Kies optie p (post)

Kies optie c (close)

Kies optie u (unlink)

P1:

Kies optie o (open semafoor)

Kies optie t (trywait)

Kies optie t (trywait)

Kies optie c (close)

Kies optie u (unlink)

Omdat het aanmaken, openen, sluiten en unlinken al in het vorige scenario is belicht zal ik nu uitsluitend het resultaat van de trywait tonen. Afbeelding 9 laat het resultaat zien van de eerste trywait. Er is hier nog geen 'lock' is dus de aanroep van de functie `sem_trywait()` heeft 0 als return gegeven. Zoals verwacht zal de tweede aanroep van trywait resulteren in een foutcode (-1). Afbeelding 10 laat hiervan het resultaat zien.

```
Enter choice: t
Calling sem_trywait(0xb7787000)
sem_trywait() returned 0
```

Afbeelding 9

```
Enter choice: t
Calling sem_trywait(0xb7787000)
ERROR/WARNING: sem_trywait() failed: Resource tempo
rarily unavailable
sem_trywait() returned -1
```

Afbeelding 10

Een ander interessant ding om te onderzoeken is hoe de waarde van de semafoor zich gedraagt. Hiervoor wil ik het volgende scenario uitvoeren:

P0:

Kies optie n (create new)

Kies optie g (getvalue)

Kies optie g (getvalue)

Kies optie g (getvalue)

Kies optie w (wait)

Kies optie w (wait)

Kies optie w (wait)

Kies optie c (close)

Kies optie u (unlink)

P1:

Kies optie o (open semafoor)

Kies optie p (post)

Kies optie p (post)

Kies optie p (post)

Kies optie c (close)

Kies optie u (unlink)

Ook nu weer zal ik de resultaten van new, open, close en unlink achterwege laten omdat deze al eerder getoond zijn.

Afbeelding 11 laat zien wat het resultaat van de eerste getvalue is. De waarde die getoond wordt is de waarde waarmee de semafoor geïnitieerd wordt.

```
Enter choice: g
Calling sem_getvalue(0xb7759000)
sem_getvalue() returned 0; value:1
```

Afbeelding 11

Vervolgens is vanuit P1 de post uitgevoerd en vanuit P0 weer de getvalue. Het resultaat is te zien in afbeelding 12. De waarde van de semafoor is dus 2 geworden.

<pre>Enter choice: g Calling sem_getvalue(0xb7759000) sem_getvalue() returned 0; value:2</pre>	<pre>Enter choice: p Calling sem_post(0xb7724000) sem_post() returned 0</pre>
--	---

Afbeelding 12

Wederom is vanuit P1 de post uitgevoerd en vanuit P0 weer de getvalue. Het resultaat is te zien in afbeelding 13. De waarde van de semafoor is dus 3 geworden.

<pre>Enter choice: g Calling sem_getvalue(0xb7759000) sem_getvalue() returned 0; value:3</pre>	<pre>Enter choice: p Calling sem_post(0xb7724000) sem_post() returned 0</pre>
--	---

Afbeelding 13

Dit betekent dus dat met deze waarde 3 processen een eventuele *critical section* in zouden kunnen gaan. Om dit aan te tonen zal er vanuit P0 nu 3x de wait worden uitgevoerd. De verwachting is dat alle 3 de keren de wait direct terug zal keren. Het uitvoeren hiervan heeft inderdaad het verwachte resultaat opgeleverd.

Om antwoord op de vraag te geven waar de semafoor zich in het filesystem bevindt, heb ik de man pages geraadpleegd⁶. Afbeelding 14 toont een listing van deze directory. Hierin is ook gelijk te zien hoe de naam van de file (semafoor) wordt gemaakt. Ik heb namelijk via het programma de naam test gebruikt. Linux maakt hier dus *sem.test* van.

```
student@student-fontys:/dev/shm$ lh
total 80K
-rwx----- 1 student student 65M mrt 14 11:48 pulse-shm-2382761758
-rwx----- 1 student student 65M mrt 3 19:10 pulse-shm-2824783943
-rwx----- 1 student student 65M mrt 3 19:10 pulse-shm-3243583467
-rw----- 1 student student 16 mrt 15 12:06 sem.test
student@student-fontys:/dev/shm$
```

Afbeelding 14

Vervolgens zijn er een aantal hexdumps gemaakt met het commando `od -A x -t x1z -v sem.test`. Hierbij is het uitgangspunt de situatie van de voorgaande opdracht. De waarde van de semafoor is dus 0. Afbeelding 15 toont het resultaat van de eerste hexdump. Hierin zien we 10 byte waarden die allen 0x00 zijn.

```
student@student-fontys:/dev/shm$ lh
total 80K
-rwx----- 1 student student 65M mrt 14 11:48 pulse-shm-2382761758
-rwx----- 1 student student 65M mrt 3 19:10 pulse-shm-2824783943
-rwx----- 1 student student 65M mrt 3 19:10 pulse-shm-3243583467
-rw----- 1 student student 16 mrt 15 12:06 sem.test
student@student-fontys:/dev/shm$ od -A x -t x1z -v sem.test
000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 >.....<
000010
student@student-fontys:/dev/shm$
```

Afbeelding 15

Na het uitvoeren van `3x` een post is er een nieuwe hexdump gemaakt. In afbeelding 16 is nu duidelijk te zien dat de hexwaarde op adres 0x000000 de waarde 0x03 heeft. Precies dus het aantal van de posts.

```
student@student-fontys:/dev/shm$ od -A x -t x1z -v sem.test
000000 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 >.....<
000010
student@student-fontys:/dev/shm$
```

Afbeelding 16

De verwachting is dat het uitvoeren van `3` waits deze waarde weer op 0 zet. Afbeelding 17 laat zien dat dit inderdaad zo is.

```
student@student-fontys:/dev/shm$ od -A x -t x1z -v sem.test
000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 >.....<
000010
student@student-fontys:/dev/shm$
```

Afbeelding 17

⁶ Accessing named semaphores via the filesystem on Linux, named semaphores are created in a virtual filesystem, normally mounted under `/dev/shm`, with names of the form `sem.somenam`. (This is the reason that semaphore names are limited to `NAME_MAX-4` rather than `NAME_MAX` characters.)

Het nogmaals uitvoeren van een wait instructie levert een verrassend resultaat. In afbeelding 18 is te zien dan niet de waarde op adres 0x000000 negatief wordt maar dat de waarde op adres 0x000008 gelijk is geworden aan 0x01.

```
student@student-fontys:/dev/shm$ od -A x -t x1z -v sem.test
000000 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 >.....<
000010
```

Afbeelding 18

Het uitvoeren van een tweede wait, vanuit P1 omdat P0 immers wacht op een lock, laat inderdaad zien dat de waarde op adres 0x000008 verhoogd is naar 0x02. Afbeelding 19 laat hiervan het resultaat zien.

```
student@student-fontys:/dev/shm$ od -A x -t x1z -v sem.test
000000 00 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00 >.....<
000010
student@student-fontys:/dev/shm$
```

Afbeelding 19

De conclusie die ik hieraan verbindt is als volgt: De interne structuur van een semafoor gebruikt twee teller variabelen.

Vervolgens moeten de volgende uitbreidingen worden gerealiseerd:

- Pas de code aan zodat de gebruiker bij optie 'n' ook kan intypen wat de initiële waarde is voor de nieuwe semafoor is.
- Pas de code aan zodat de gebruiker bij optie 'n' ook kan intypen wat de permissions voor de group en de other worden.

Voor de eerste aanpassing is de rood gemarkeerde code uit afbeelding 20 toegevoegd. Omdat het programma alleen bedoeld is om met semaforen te spelen is er bewust geen validatie op de input gedaan.

```
80      case 'n':
81          if (semdev != SEM_FAILED)
82          {
83              printf ("ERROR: another semaphore already opened\n");
84              break;
85          }
86          printf ("Enter name: ");
87          fgets (sem_name, sizeof (sem_name), stdin);
88          remove_nl (sem_name);
89
90          printf ("Enter init value :");
91          fgets (line, sizeof (line), stdin);
92          initval = atoi(line);
93
94          urights = decimal_octal(600);
95
96          printf ("Calling sem_open('%s', 0_CREAT | 0_EXCL, 0600, %d)\n", sem_name, initval);
97          semdev = sem_open (sem_name, 0_CREAT | 0_EXCL, 0666, initval);
98
99          if (semdev == SEM_FAILED)
100          {
101              perror ("ERROR: sem_open() failed");
102          }
103          printf ("sem_open() returned %#x\n", (unsigned int)semdev);
104          break;
```

Afbeelding 20

De tweede aanpassing was wat lastiger. In de opdracht werd er al op gehint dat dit met de default permissies te maken heeft zoals die middels *UMASK* ingesteld zijn. Het opvragen van de default umask levert op dat deze 0022 is.

In Linux wordt umask gebruikt voor het aanmaken van nieuwe files en directories. In mijn geval zou het aanmaken van een nieuwe file met de permissie 666 resulteren in 644. Dit betekent dus dat de opgegeven permissie (666) gemaskeerd wordt met het inverse bitmasker van 0022. Voor het maskeren wordt een logische AND gebruikt.

Als we even alleen de 6 voor de permissie en de 2 van de umask onder elkaar zetten dan wordt

6 = 0110

2 = 0010 => ~0010 = 1101

0110

1101

----- &

0100

0100 is gelijk aan 4. Vandaar dat 666 resulteert in permissie 644.

Om dit (tijdelijk) te ondervangen heb ik in de shell het commando *umask 0* gegeven. Hiermee is het volledige masker gereset en heb ik volledige vrijheid bij het creëren van een file.

Om de gebruiker om de gewenste file permissies te vragen is er in het programma een aantal aanpassingen gedaan. Als eerste is er een functie toegevoegd om octaal om te rekenen naar decimaal. Afbeelding 21 laat hiervan de implementatie zien. Deze functie is nodig zodat de gebruiker bijvoorbeeld 600 of 666 kan invoeren. Aangezien dit octale getallen zijn moeten die dus worden omgerekend. Vervolgens zijn in het programma de aanpassingen gemaakt zoals die in het rode kader zijn aangegeven in afbeelding .

```
38 int
39 octal_decimal(int n)
40 {
41     int decimal=0, i=0, rem;
42
43     while (n != 0)
44     {
45         rem = n%10;
46         n /= 10;
47         decimal += rem * pow(8,i);
48         i++;
49     }
50     return decimal;
51 }
```

Afbeelding 21

```
96
97     printf("Enter init value :");
98     fgets(line, sizeof(line), stdin);
99     initval = atoi(line);
100
101     printf("Enter permissions (Owner, Group, Other) i.e. 666 :");
102     fgets(line, sizeof(line), stdin);
103     urights = atoi(line);
104
105     printf("Calling sem_open('%s', 0_CREAT | 0_EXCL, 0_RDWR, %s)\n", sem_name, urights, initval);
106     semdes = sem_open(sem_name, 0_CREAT | 0_EXCL, 0_RDWR, octal_decimal(urights), initval);
107
108     if (semdes == SEM_FAILED)
```

Afbeelding 22

Nog even een klein stukje theoretische opfrissing:

Het talstelsel met het grondtal of radix = 8 noemen we het octale stelsel. Hiervoor geldt:

- De symbolen zijn: 0, 1, 2, 3, 4, 5, 6, 7.
- De posities betekenen: ... 8^4 , 8^3 , 8^2 , 8^1 , 8^0 .

Omrekenen van een octaal naar een decimaal getal gaat dus als volgt:

$$\begin{array}{rcl} 642 \text{ (octaal)} & 2 \times 8^0 = & 2 \\ & 4 \times 8^1 = & 32 \\ & 6 \times 8^2 = & 384 \\ \hline & & 418 \text{ (decimaal)} \end{array}$$

Als nu het programma wordt gestart en er wordt een nieuwe semafoor aangemaakt dan kan deze inderdaad geïnitieerd worden met de opgegeven parameter en is het ook mogelijk om de permissie in te stellen. Afbeelding 23 toont de permissie en afbeelding 24 toont een hexdump met de opgegeven init waarde (0x0A = 10) voor de semafoor.

```
student@student-fontys:/dev/shm$ ls -l
total 80K
-rwx----- 1 student student 65M mrt 14 11:48 pulse-shm-2382761758
-rwx----- 1 student student 65M mrt  3 19:10 pulse-shm-2824783943
-rwx----- 1 student student 65M mrt  3 19:10 pulse-shm-3243583467
-rw-rw-rw- 1 student student 16 mrt 15 16:52 sem.test
student@student-fontys:/dev/shm$
```

Afbeelding 23

```
student@student-fontys:/dev/shm$ od -A x -t x1z -v sem.test
000000 0a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 >.....<
000010
student@student-fontys:/dev/shm$
```

Afbeelding 24

Antwoord op de vragen (2.2 Synchronisatie)

Gevraagd worden 4 processen A, B, C, D (dus geen threads!) die het volgende herhaaldelijk uitvoeren: Ze drukken de getallen 1 t/m 8 af op een gemeenschappelijke terminal; Hierbij drukt proces A de getallen 1 en 5 af; proces B de getallen 2 en 6; proces C de getallen 3 en 7; proces D de getallen 4 en 8.

Eisen:

- De getallen worden in de "goede volgorde" afgedrukt
- Er mogen alleen semaforen gebruikt worden om te synchroniseren (dus geen busy-wait lussen en geen shared variabelen).
- Het mag geen verschil uitmaken welk proces als eerste wordt opgestart (maar het is wel toegestaan om de semaforen van te voren al te creëren, bijv. met een ander proces)

Uitwerking:

Voor ieder proces zal op voorhand een semafoor worden aangemaakt. Voor proces A zal deze worden geïnitieerd op 1 en voor proces B, C en D zal dit 0 zijn. Verder zal in pseudo code een proces er als volgt uitzien:

PA:	PB:	PC:	PD:
While(1) {	While(1) {	While(1) {	while(1) {
wait(A)	wait(B)	wait(C)	wait(D)
print(1)	print(2)	print(3)	print(4)
post(B)	post(C)	post(D)	post(A)
wait(A)	wait(B)	wait(C)	wait(D)
print(5)	print(6)	print(7)	print(8)
post(B)	post(C)	post(D)	post(A)
}	}	}	}

Afbeelding 25 laat de main loop zien van de gerealiseerde code. Hierbij wil ik de kanttekening maken dat er in de loop geen foutafhandeling plaats vindt. Ik heb hiervoor bewust gekozen om zo het clean mogelijk te houden waardoor het makkelijker is om het concept te zien. Uitvoeren van de processen in een willekeurige volgorde levert altijd het gewenste resultaat.

```
117 int
118 main(int argc, char * argv[])
119 {
120
121     int        initresult;
122
123     if (argc != 2)
124     {
125         fprintf (stderr, "Usage: %s <proces-number 0..4>\n", argv [0]);
126         return (-1);
127     }
128
129     initresult = init_params(atoi(argv[1]));
130
131     if(initresult < 0)
132     {
133         perror("Initialize fatal error");
134         return(-1);
135     }
136
137     while(1)
138     {
139         sem_wait (this_semdes);
140         printf("%c\n", prnchars[0]);
141         sem_post(next_semdes);
142         sem_wait(this_semdes);
143         printf("%c\n", prnchars[1]);
144         sem_post(next_semdes);
145     }
146 }
```

Afbeelding 25

Antwoord op de vragen (2.3 rendez-vous)

Voor deze opdracht hebben we vier processen die allen twee statements (statement_1 en statement_2) uit willen voeren.

Schrijf één programma dat je vier keer tegelijkertijd opstart zodat er vier processen draaien. Elk proces mag zijn statement_2 pas uitvoeren nadat alle vier de processen hun eerste statement_1 hebben uitgevoerd. Alle statement_1's moeten door elkaar uitgevoerd kunnen worden, en dat geldt ook voor alle statement_2's.

De code ziet er zoiets uit als:

```
initialisatie  
statement_1  
sync_code  
statement_2
```

De synchronisatie (sync_code) moet met semaphores en shared memory plaatsvinden; niet met busy-wait lussen (het is overigens ook mogelijk om uitsluitend semaphores te gebruiken (zonder shared memory)).

Uitwerking:

Ieder proces moet weten wanneer het zelf zijn statement_1 heeft uitgevoerd, maar moet dit ook van de andere drie processen weten. Om dit bij te kunnen houden zal van een byte de 4 LSB worden gebruikt. Bit 0 houdt bij of proces 1 zijn statement_1 heeft uitgevoerd, bit 1 is doet dit voor proces 2 enz. Ieder van deze 4 bits is dus een vlag die uniek aangeeft of het statement_1 voor een proces is uitgevoerd. De processen elkaars informatie nodig waardoor het nodig is om dit in een shared memory onder te brengen. Omdat mutual exclusion bij het lezen en schrijven van deze byte van belang is zal dit worden beschermt door een gezamenlijke semafoor.

Voor de synchronisatie van de vier processen is er is feitelijk maar één moment in de afloop van belang. Dit is het moment dat alle vier de processen hun statement_1 hebben uitgevoerd en dus ieder proces weer verder mag met de executie van statement_2. De volgorde van executie van statement_2 is hierbij niet van belang waardoor het ook niet van belang is om verder nog bij te houden welk proces statement_2 heeft uitgevoerd. Om dit te realiseren zal er gebruik worden gemaakt van een tweede gezamenlijk semafoor. Deze semafoor zal op 0 worden geïnitieerd.

De volgende pseudo code zal het geheel verduidelijken:

sem_flags = 1, sem_sync = 0.

```
1:   execute statement_1
2:   (* enter cs *)
3:   sem_wait(sem_flags)
4:   read status flags from shm into variable
5:   mask variable with flag
6:   update shm with variable
7:   (* leave cs *)
8:   sem_post(sem_flags)
9:   if(all flags are set)
10:      sem_post(sem_sync)
11:   else
12:      sem_wait(sem_sync)
13:   sem_post(sem_sync)
14:   execute statement_2
```

Afbeelding 26 toont het resultaat als de processen in een verschillende volgorde worden opgestart. Hierin is duidelijk te zien dat de processen op elkaar wachten voordat ze hun statement_2 uitvoeren. Hierbij moet wel aangemerkt worden dat proces 0 altijd als eerste gestart moet worden. Proces 0 initialiseert de semaforen en de shared memory.

```
student@student-fontys:~/bart/programming/school/opdracht.2/IPC32_ipc$  
Stament 1 from proces 0  
Stament 1 from proces 2  
Stament 1 from proces 1  
Stament 1 from proces 3  
Stament 2 from proces 0  
Stament 2 from proces 2  
Stament 2 from proces 1  
Stament 2 from proces 3  
  
student@student-fontys:~/bart/programming/school/opdracht.2/IPC32_ipc$  
Stament 1 from proces 0  
Stament 1 from proces 3  
Stament 1 from proces 1  
Stament 1 from proces 2  
Stament 2 from proces 0  
Stament 2 from proces 3  
Stament 2 from proces 1  
Stament 2 from proces 2  
  
student@student-fontys:~/bart/programming/school/opdracht.2/IPC32_ipc$ █
```

Afbeelding 26