OSM, G4

Anders Iversen, Mikkel Mastek, Anders Bjerg Pedersen

4. marts 2008

G4.1 Pipe i delt hukommelse

Implementationen af `pipe.c` genbruger en hel del kode fra `shm.c` til at oprette den fælles del af hukommelsen og semaforen. I headeren er desuden inkluderet de relevante biblioteker. Herunder beskrives kort, hvad der er gjort i de forskellige dele.

pipe_t:
Her er blot tilføjet vores semafor med `sem_t sem`.

pipe_create:
I stedet for at malloc'e vores `pipe_t`-objekt allokerer vi nu objektet i et stykke fælles hukommelse, på samme måde som i `shm.c`, og lader vores `pipe_t` pege på den delte hukommelse.

pipe_free:
I stedet for at anvende kommandoen `free()`, anvender vi `shmdt()` til at frigøre del delte hukommelse.

pipe_write:
Indsat en `sem_wait()` før vi gør noget som helst, og en `sem_post()` når vi er færdige for at beskytte adgangen til vores pipe.

pipe_read:
Indsat en `sem_wait()` før vi gør noget som helst, og en `sem_post()` når vi er færdige for at beskytte adgangen til vores pipe.

print_pipe():
Denne funktion er tilføjet for at "servicere brugeren"og kalder `pipe\_read()` og udskriver resultatet til skærmen.

Med hensyn til testen udføres denne i `main`-proceduren. Vi forker en child-proces, der løbende skal skrive en længere tekststreng ind i bufferen, og hvor parent-processen så gradvist skal udskrive indholdet af bufferen. Dette er igen implementeret som i `shm.c`, hvor begge processer i et while-loop forsætter med at læse/skrive, indtil hele tekststrengen er blevet indlæst og udskrevet igen. Eftersom læse/skrive-funktionerne anvender den samme semafor, skulle processerne gerne ping-ponge mellem hinanden. Ligeledes har vi ladet parent-processen udtage et tilfældigt antal karakterer ad gangen (så outputtet ikke

bliver alt for trivielt...). Til sidst frigiver begge processer den delte hukommelse.

I nedenstående output ses det, at hele tekststrengen bliver udskrevet af parent-processen (`pid: 11468`) i bidder af tilfældig længde. Det ses også, at parent-processen 3 gange forsøger at hente data fra en tom buffer eller hente 0 karakterer (`len=0`). Ved gentagne kørsler med forskellig pipe-størrelse har vi bemærket, at parent-processen får langt mere processortid end dens child-proces. Hvad dette skyldes, er for os ukendt.

```
bach-3 > ./pipe 10
pid: 11468, len=2: Je
pid: 11468, len=8: g er hav
pid: 11468, len=1: r
pid: 11468, len=1: e
pid: 11468, len=0:
pid: 11468, len=0:
pid: 11468, len=0:
pid: 11468, len=4: n, j
pid: 11468, len=7: eg har
pid: 11468, len=2: bj
pid: 11468, len=6: ælder
pid: 11468, len=5: på...
pid: 11468, len=6:  Bliv
pid: 11468, len=8: spejder
pid: 11468, len=6: min dr
pid: 11468, len=4: eng!
bach-3 >
```

G4.2 Segmenterede sidetabeller

Vi begynder med at omskrive selectoren til binær form og afkoder derefter segment number (s) til decimalform. Ligeledes omskriver vi offset til binær notation:

Selector:	**Selector** (binær, *s-g-p*):	s (dec.)	**offset:**	**offset** (binær):
0x270	0001001110 0 00	78	0x11	00000000000000000000000000010001
0x278	0001001111 0 00	79	0xF0	00000000000000000000000011110000
0x10C	0000100001 1 00	33	0xF0F	00000000000000000000111100001111

Vi ser ud fra værdierne af g, at de to første skal slås op i GDT og den sidste skal slås op i LDT. Derefter tjekker vi limit op mod offset og ser, at nummer to giver en hukommelsesfejl (offset > limit), der resulterer i en trap til styresystemet. For de to andre beregner vi lineær adresse som base + offset:

Base:	**Offset:**	**Lineær adresse** (base + offset):				**Limit:**	**Trap:**
0x803000	0x11	0x803011 =	0000000010	0000000011	000000010001	0x100	-
			p_1=2	p_2=3	d=0x11		
0xC01000	0xF0	0xC010F0=	0000000011	0000000001	000011110000	0x80	Trap
			p_1=3	p_2=1	d=0xF0		
0xC00000	0xF0F	0xC00F0F=	0000000011	0000000000	111100001111	0x1000	-
			p_1=3	p_2=0	d=0xF0F		

Ud fra den lineære adresse får vi p_1, der bruges til opslag i sidekataloget, p_2 der bruges til opslag i sidetabellerne samt d, der adderet med opslaget på den fundne sideplads giver os den fysiske adresse:

Selector:	d (offset):	**I sideplads:**	**Fysisk adresse:**
0x270	0x11	0x07	0x11+0x07=0x18=00000000000000000000000000011000
0x278	0xF0	-	-
0x10C	0xF0F	0x43	0xF0F+0x43=0xF52=00000000000000000000111101010010