



IT-Universitetet  
i København

OBLIGATORISK OPGAVE # 3 I BOSC

---

## Operativsystemer og C

---

*Author:*

Tom Mørk Christensen

Jonas Elbækgaard Jørgensen

*ITU-mail:*

TMCH@ITU.DK

JELB@ITU.DK

7. november 2014

# Indhold

<b>1</b>	<b>Introduktion</b>	<b>2</b>
<b>2</b>	<b>Metode</b>	<b>2</b>
<b>3</b>	<b>Implementation</b>	<b>3</b>
3.1	Linux Kernel Modules . . . . .	3
3.2	Linux Kernel Module for Listing Tasks . . . . .	3
<b>4</b>	<b>Test</b>	<b>4</b>
4.1	Linux Kernel Modules . . . . .	4
4.2	Linux Kernel Module for Listing Tasks . . . . .	4
<b>5</b>	<b>Diskussion</b>	<b>5</b>
5.1	Linux Kernel Module for Listing Tasks . . . . .	5
<b>6</b>	<b>Konklusion</b>	<b>5</b>
<b>A</b>	<b>Appendiks</b>	<b>6</b>
A.1	Linux Kernel Modules . . . . .	6
A.2	Linux Kernel Module for Listing Tasks - del 1 . . . . .	7
A.3	Linux Kernel Module for Listing Tasks - del 2 . . . . .	8

# 1 Introduktion

I forbindelse med udarbejdelsen af denne rapport har vi arbejdet med kerne moduler til operativsystemet Linux. Arbejdet med modulerne har været opdelt i to hovedopgaver.

I den første del ”*Linux Kernel Modules*” [1, s.94] har vi beskæftiget os med udarbejdelsen af et simpelt modul for at undersøge hvordan disse skal udformes samt for at undersøge hvordan moduler tilføjes til kernen samt hvordan de fjernes igen efter brug. Her udover har denne del af opgaven også fungeret platform for at få indsigt i hvordan en række makroer fungerer.

Anden halvdel af den obligatoriske opgave er at løse Project 2 - Linux Kernel Module for Listing Tasks [1, s.156-158]. Opgaven går ud på at skrive et linux kernemodul, som kan skrive til kernel log bufferen hvilke processer operativsystemet kører ved modulets indlæsningstidspunkt.

Delopgaven er yderligere opdelt i 2 dele, og til hver del hører forskellige krav.

**Del 1:** Skab et modul, som itererer igennem alle tasks i systemet. Skriv taskens navn, ID og state til loggen. De er ikke påkrævet at skrive eventuelle barneprocesser til disse tasks.

**Del 2:** Skab et modul, som iterer igennem alle tasks i systemet, samt deres barneprocesser, ved brug af en dybde-først algoritme.

## 2 Metode

### Allokering og frigivelse af hukommelse

Til at allokere plads i kernes hukommelse benytter vi os af funktionen `kmalloc`. Som det gør sig gældende for den hyppigere anvendte `malloc` returnerer `kmalloc` også en pointer til en adresse i hukommelsen hvor der er reserveret det efterspurgte antal bytes. Når vi ønsker at frigive hukommelsen igen gør vi brug af `kfree` som frigiver den allokerede hukommelse.

### Logning af data

Når vi ønsker at skrive informationer ud til brugeren af vores modul gør vi brug af system kaldet `printk`, som kan ses som værende den kerne specifikke version af funktionen `printf`. For begge funktioner gør det sig gældende at det er muligt at formatere den resulterende tekststreng ved brug af diverse parameter. I forhold til `printf` benytter `printk` sig af et såkaldt ”*log level*” til at specificere vigtigheden af en given log besked. I vores implementation gør vi brug af log niveauet `KERN_INFO`, som indikere at beskeden indeholder simple information. Linux kernen understøtter i alt 8 forskellige log niveauer som dækker fra uskyldige debug beskeder, `KERN_DEBUG` til nødsituationer af typen `KERN_EMERG`.

For at læse indholdet af kernens buffer gør vi brug af `dmesg` kommandoen, som skriver indholdet af bufferen til terminalen.

### Iterering over lister

Til at iterere over elementer i lister gør vi brug af makroerne `list_for_each` og `list_for_each_entry_safe`. Grunden til at vi også bruger safe versionen af denne makro er at den tillader at der slettes elementer fra listen under iterationen. Dette skyldes at safe versionen tager en ekstra pointer, af samme type som elementerne. Dette parameter bruges som midlertidig lager. Dette lager bruges til at gemme det næste element i listen inden der ændres på det aktuelle element. Når alle ændringer af elementet er udført bliver elementet i det midlertidige lager

til det aktuelle element og dets efterfølger gemmes i lageret. På denne måde er referancen til listens næste element altid bevaret uanset om det aktuelle element slettes. I modsætning gør makroen `list_for_each_entry` ikke brug af et midlertidigt lager og referancen til det næste element vil derfor gå tabt hvis det aktuelle element fjernes fra listen.

## 3 Implementation

### 3.1 Linux Kernel Modules

Vores implementation er baseret på en løkke som ved hver iteration opretter én instans af `birthday` strukturen og tilføjer denne til enden af modulets liste `birthday_list`.<sup>1</sup>

Samtidig med at modulets initialiserings funktion `void simple_init(void)` opretter `birthday` strukturerne skrives der information om de enkelte instanser til kernens buffer. `Birthday` strukturen består af fire felter, `int day`, `int month`, `int year` og `list_head list`. De tre heltal(`int`) bruges til at representere en date, mens `list` bruges hængte listens elementer sammen. `list_head` er kernens implementation af en dobbelt hængtet liste, hvor en instans holder to pegere, en til det forgående element, og en til det efterfølgende element. For at tilføje en `birthday` struktur til listen bruger vi makroen `list_add_tail`, som tilføjer et element bagerst til listen.

I modulets exit funktion `void simple_exit(void)` benytter makroen `list_for_each_entry_safe` til at iterere over elementerne i listen og skriver information om de enkelte elementer til logge før vi sletter dem ved hjælp af funktionen `kfree`.

### 3.2 Linux Kernel Module for Listing Tasks

I Linux er processer organiseret som tasks af typen `struct task_struct`. Fra denne struct kan vi tilgå taskens navn, state og id, og udskrive dem til buffer loggen. Structen indeholder også macroen `for_each_process(struct task_struct*)` som iterere over alle igangværende tasks.

Structen indeholder desuden en pointer `struct list_head` til en liste over den egne barneprocesser.

**Del 1:** I filen `taskPrinter.c` opretter vi først en `struct task_struct` pointer og bruge denne som input i macroen. Herefter indsættes en `printk(KERN_INFO "...")` linie i macroen. Dette giver os et udskrift af den ønskede information fra hver task, men ikke for eventuelle barneprocesser.<sup>2</sup>

**Del 2:** I filen `taskAndChildPrinter.c` bygger vi videre på indholdet af `taskPrinter.c`. Da vi også ønsker at udskrive barneprocesserne, tilføjer vi en variabel `int generation`: denne variabel vil vi bruge til at tælle hvor dybt i træstrukturen en proces befinder sig.<sup>3</sup>

Herefter tilføjer vi en metode `dfs(struct task_struct *parentTask, int generation)` som skal kaldes rekursivt. I denne metode bruger vi liste-macroen `list_for_each(list, &init_list)` til at iterere over alle barneprocesser i en enkelt forældreproces. I hver iteration skrives de ønskede informationer til loggen, og `dfs` metoden kaldes på ny. Denne rekursion fortsætter således med at dykke et niveau ned indtil den når DFS-træets blade, som beskrevet

---

<sup>1</sup>Se appendix A.1 for kildekode.

<sup>2</sup>Se appendix A.2 for kildekode.

<sup>3</sup>Se appendix A.3 for kildekode.

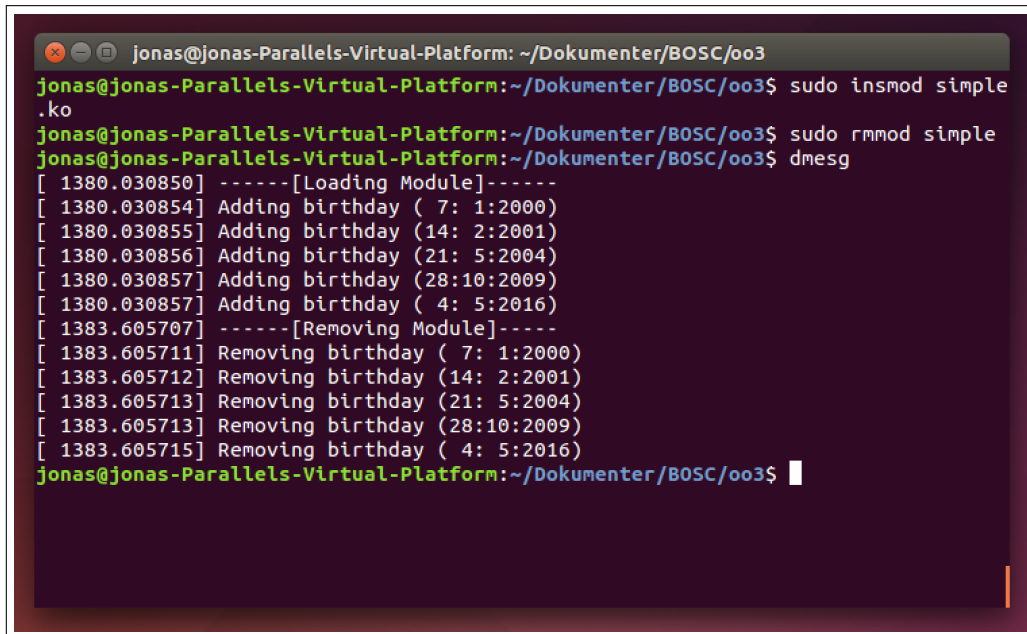
i [1, s.157, s.114].

Udover at udskrive navn, nummer og state for en given proces eller barneprocess, skriver vi også ID nummer på den umiddelbare forældreproces, samt hvilken generation der er tale om: processer på højeste niveau (tasks) har generation 0, deres børn har generation 1 og så fremdeles.

## 4 Test

### 4.1 Linux Kernel Modules

Vi har kun udført en test af vores kerne modul. Denne test bygger på hvorvidt modulet kan installeres og afinstalleres. I samme forbindelse testes der også at modulet er i stand til at skrive til kernens buffer. Figur 1 viser udskrift i terminalen af beskeder gemt i kernens buffer fra indstættelse til fjernelse af modulet.



```
jonas@jonas-Parallels-Virtual-Platform: ~/Dokumenter/BOSC/oo3
jonas@jonas-Parallels-Virtual-Platform:~/Dokumenter/BOSC/oo3$ sudo insmod simple
.ko
jonas@jonas-Parallels-Virtual-Platform:~/Dokumenter/BOSC/oo3$ sudo rmmod simple
jonas@jonas-Parallels-Virtual-Platform:~/Dokumenter/BOSC/oo3$ dmesg
[ 1380.030850] -----[Loading Module]-----
[ 1380.030854] Adding birthday ( 7: 1:2000)
[ 1380.030855] Adding birthday (14: 2:2001)
[ 1380.030856] Adding birthday (21: 5:2004)
[ 1380.030857] Adding birthday (28:10:2009)
[ 1380.030857] Adding birthday ( 4: 5:2016)
[ 1383.605707] -----[Removing Module]-----
[ 1383.605711] Removing birthday ( 7: 1:2000)
[ 1383.605712] Removing birthday (14: 2:2001)
[ 1383.605713] Removing birthday (21: 5:2004)
[ 1383.605713] Removing birthday (28:10:2009)
[ 1383.605715] Removing birthday ( 4: 5:2016)
jonas@jonas-Parallels-Virtual-Platform:~/Dokumenter/BOSC/oo3$
```

Figur 1: Print af kernens buffer til terminalen.

### 4.2 Linux Kernel Module for Listing Tasks

Silberschatz foreslår at man afprøver sin løsning ved i kommandolinien at kalde `ps -e1` og `ps -eLf`. Disse systemkald returnerer hhv. alle nuværende tasks og alle nuværende tasks plus barneprocesser. Derudover skriver han også at tasks er dynamiske, og listerne kan afvige fra vore egne udskrift. Ved at køre de to systemkald får vi dog resultater som umiddelbart lader til at stemme overens med de resultater vores løsning giver.

Udover denne metode har vi implementeret en ganske simpel tæller, som i løbet af iterationen summerer hvor mange aktive processer der observeres, og udskriver resultatet til loggen. Resultatet af disse summeringer har ligget omkring 157 for tasks alene og 531 på samtlige processer. Linierne med summeringskoden er udkommenteret i afleveringen.

Til sidst har vi udvidet den printede information til at indeholde en proces' forældres ID nummer, samt processens generations nummer. Dette betyder, at det er muligt for en given 'procesfamilie' at kontrollere i hvilken rækkefølge processerne har avlet hinanden, og se at de enkelte processer skrives ud i den påkrævede orden.

## 5 Diskussion

### 5.1 Linux Kernel Module for Listing Tasks

Opgaven er i sig selv uhyre simpel, så længe man som programmør har beskæftiget sig blot en smule med træ-datastrukturer og DFS/BFS søgning. Udfordringen har været at implementere en løsning i et sprog som stadig føles lidt uvant, på en datastruktur som er flere hundrede linier lang og placeret i en flere tusind linier stor fil, ved hjælp af macroer hvis anvendelse ikke er dokumenteret, på et datasæt som kan ændre sig mellem kørsler. Det har altså været svært at omsætte den lette opgavebeskrivelse til en løsning.

Ud fra resultaterne af vore eksperimenter, både med optællingen, systemkald og ved at analysere de faktiske udskrift til loggen, mener vi at kunne sige at vores løsning lever op til kravene. Men uden dybere kendskab processerne og hvordan vores specifikke Linuxversion benytter dem, er det svært at sige med absolut sikkerhed at alting opfører sig som det skal.

En ting vi har observeret som har betydet at vi har tvivlet på løsningens korrekthed, er tilfælde hvor en proces med samme navn og ID dukker op flere gange, nogle gange som task, og andre gange som barneprocess. Vi er ikke klar over om dette er helt naturligt; at en proces kan være barn af en task, og som resultat deraf selv blive en task, eller der er en fundamental fejl i vores forståelse af tasks og processers natur.

## 6 Konklusion

### Litteratur

- [1] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system concepts*. John Wiley & Sons, 9<sup>th</sup> edition, 2013.

# A Appendiks

## A.1 Linux Kernel Modules

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3 #include <linux/kernel.h>
4 #include <linux/slab.h>
5
6 struct birthday {
7     int day;
8     int month;
9     int year;
10    struct list_head list;
11 };
12
13 static LIST_HEAD(birthday_list);
14
15 void create(int day, int month, int year, struct birthday *person);
16
17 int simple_init(void)
18 {
19     struct birthday *person;
20     int i;
21     printk(KERN_INFO "-----[Loading Module]-----\n");
22
23     for (i = 0; i < 5; i++) {
24         person = kmalloc(sizeof(*person), GFP_KERNEL);
25         person->day = ((i+1)*100)%31;
26         person->month = ((i*i)%12)+1;
27         person->year = 2000 + i*i;
28         INIT_LIST_HEAD(&person->list);
29         printk(KERN_INFO "Adding birthday (%2d:%2d:%4d)\n", person->day, person->
30             month, person->year);
31         /* ADDING ELEMENT TO LIST */
32         list_add_tail(&person->list, &birthday_list);
33     }
34     return 0;
35 }
36
37 void simple_exit(void) {
38     struct birthday *ptr, *next;
39     printk(KERN_INFO "-----[Removing Module]-----\n");
40
41     list_for_each_entry_safe(ptr, next, &birthday_list, list) {
42         printk(KERN_INFO "Removing birthday (%2d:%2d:%4d)\n", ptr->day, ptr->month
43             , ptr->year);
44         list_del(&ptr->list);
45         kfree(ptr);
46     }
47 }
48
49 module_init( simple_init );
50 module_exit( simple_exit );
51
52 MODULE_LICENSE("GPL");
53 MODULE_DESCRIPTION("Simple Module");
54 MODULE_AUTHOR("SGG");
```

Listing 1: Implementation af simpelt modul - *simple.c*

## A.2 Linux Kernel Module for Listing Tasks - del 1

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3 #include <linux/kernel.h>
4 #include <linux/slab.h>
5 #include <linux/sched.h>
6
7 //int count;
8
9 int taskprinter_entry(void)
10 {
11     struct task_struct *task;
12     //count = 0;
13     printk(KERN_INFO "ID:      Name:      State:");
14     for_each_process(task) {
15         //count = count + 1;
16         printk( KERN_INFO "%6i%20s%10ld\n", task->pid, task->comm, task->state);
17     }
18     //printk(KERN_INFO "Processes count: %i\n", count);
19     return 0;
20 }
21
22
23 void taskprinter_exit(void)
24 {
25     printk(KERN_INFO "-----[Removing Module]-----\n");
26 }
27
28
29 module_init( taskprinter_entry );
30 module_exit( taskprinter_exit );
31
32 MODULE_LICENSE("GPL");
33 MODULE_DESCRIPTION("Task Printer");
34 MODULE_AUTHOR("SGG");
```

Listing 2: Implementation af simple task printer - *taskPrinter.c*



### A.3 Linux Kernel Module for Listing Tasks - del 2

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3 #include <linux/kernel.h>
4 #include <linux/slab.h>
5 #include <linux/sched.h>
6
7 //int count;
8
9 void dfs(struct task_struct *parentTask, int generation)
10 {
11     struct list_head *list;
12     generation = generation + 1;
13
14     list_for_each(list, &parentTask->children)
15     {
16         struct task_struct *childTask;
17         //count = count + 1;
18         childTask = list_entry(list, struct task_struct, sibling);
19         printk(KERN_INFO "%6i%9i%5i%19s%7ld\n", childTask->pid, parentTask->pid,
20             generation, childTask->comm, childTask->state);
21         dfs(childTask, generation);
22     }
23 }
24
25 int taskprinter_entry(void)
26 {
27     struct task_struct *task;
28     int generation;
29     generation = 0;
30     //count = 0;
31
32     for_each_process(task)
33     {
34         //count = count + 1;
35         printk(KERN_INFO "      ID  PARENT  GEN      NAME  STATE");
36         printk(KERN_INFO "%6i%9s%5i%19s%7ld\n", task->pid, "N/A", generation, task
37             ->comm, task->state);
38         dfs(task, generation);
39         printk(KERN_INFO "-----\n");
40     }
41     //printk(KERN_INFO "Processes count: %i\n", count);
42     return 0;
43 }
44
45 void taskprinter_exit(void)
46 {
47     printk(KERN_INFO "-----[Removing Module]-----\n");
48 }
49
50 module_init( taskprinter_entry );
51 module_exit( taskprinter_exit );
52
53 MODULE_LICENSE("GPL");
54 MODULE_DESCRIPTION("Task Printer");
55 MODULE_AUTHOR("SGG");
```

Listing 3: Implementation of DFS task printer - *taskAndChildPrinter.c*