

Musterlösungen zu den Tutoraufgaben auf
Blatt 2
der Übungen zur Vorlesung
“Grundlagen Betriebssysteme und Systemsoftware

M.Kramm, G.Groh, 31.10.2008

- 1.** An operating system must provide the users with an extended (i.e., virtual) machine, and it must manage the I/O devices and other system resources.
- 2.** Multiprogramming is the rapid switching of the CPU between multiple processes in memory. It is commonly used to keep the CPU busy while one or more processes are doing I/O.
- 3.** Input spooling is the technique of reading in jobs, for example, from cards, onto the disk, so that when the currently executing processes are finished, there will be work waiting for the CPU. Output spooling consists of first copying printable files to disk before printing them, rather than printing directly as the output is generated. Input spooling on a personal computer is not very likely, but output spooling is.

4. The prime reason for multiprogramming is to give the CPU something to do while waiting for I/O to complete. If there is no DMA, the CPU is fully occupied doing I/O, so there is nothing to be gained (at least in terms of CPU utilization) by multiprogramming. No matter how much I/O a program does, the CPU will be 100 percent busy. This of course assumes the major delay is the wait while data are copied. A CPU could do other work if the I/O were slow for other reasons (arriving on a serial line, for instance).

8. Choices (a), (c), and (d) should be restricted to kernel mode.

10. Every nanosecond one instruction emerges from the pipeline. This means the machine is executing 1 billion instructions per second. It does not matter at all how many stages the pipeline has. A 10-stage pipeline with 1 nsec per stage would also execute 1 billion instructions per second. All that matters is how often a finished instructions pops out the end of the pipeline.

Text im Blatt:

"Ganz interessant ist es, in die Assembler-Versionen (.s-Files) der Kompilate der C-Programme reinzuschauen. Mit Hilfe des Tools `objdump` können außerdem weitere Einblicke in die Struktur der Binaries und in die Arbeitsweise des Linkers gewonnen werden."

Frage:

Woran erkennt man in der Ausgabe von `objdump main4a` bzw. `objdump main4b`, dass `say_hello()` einmal statisch gelinkt ist und einmal nicht?

Antwort:

In `objdump -DR -d main4a` (`main4a` ist statisch gegen `hellworld.o` gelinkt) sieht man folgenden Aufruf:

```
call    80483d4 <say_hello>
```

Zur Hausaufgabe 1.6: Bemerkungen

Die Adresse (hier) 80483d4 kommt hier desweiteren tatsächlich in der objdump Ausgabe vor, und definiert die Funktion `say_hello`:

```
080483d4 <say_hello>:  
80483d4: 55          push    %ebp  
80483d5: 89 e5      mov     %esp,%ebp  
...
```

D.h. in `main4a` springt der Funktionsaufruf direkt an eine existierende Funktion im selben Executable.

In `objdump -DR main4b` (`main4b` ist dynamisch gegen `helloworld.o` gelinkt) hingegen sieht die `call` Anweisung folgendermaßen aus:

```
call    80483dc <say_hello@plt>
```

Adresse 80483dc enthält desweiteren folgenden Code:

```
080483dc <say_hello@plt>:  
80483dc: ff 25 04 a0 04 08      jmp     *0x804a004
```

Desweiteren liefert ein Blick in den GLOBAL OFFSET TABLE (etwas weiter unten):

```
0804a004: R_386_JUMP_SLOT          say_hello
```

D.h. hier wird der Funktionsaufruf an einen Jump weitergereicht, der wiederum in die vom Linker erzeugte Sprungtabelle springt.

Text im Blatt:

"Bauen Sie z.B. zusätzlich in C-Programme Endlosschleifen ein und schauen Sie sich mit Hilfe der Process-ID die Inhalte der zugehörigen `proc`-Einträge an."

Vorgehen:

`helloworld.c` folgendermassen abändern:

```
void say_hello()
{
    while(1) {}
}
```

Danach nochmals `make` aufrufen, und sowohl `main4a` als auch zwei `main4b` starten:

```
LD_LIBRARY_PATH=. ./main4a &
LD_LIBRARY_PATH=. ./main4b &
LD_LIBRARY_PATH=. ./main4b &
```

Zur Hausaufgabe 1.6: Bemerkungen

Ein

```
ps | grep main
```

liefert z.B.:

```
25417 pts/6      00:00:09 main4a
25418 pts/6      00:00:07 main4b
25419 pts/6      00:00:07 main4b
```

Für die drei Prozess-IDs sieht man sich jetzt in `/proc/<ProzessID>/maps` an:

```
cat /proc/25417/maps
```

```
...
(*kein* libhelloworld.so zu sehen)
...
```

```
cat /proc/25418/maps
```

```
...
b7fad000-b7fae000 r-xp 00000000 08:04 6587284 libhelloworld.so
b7fae000-b7faf000 r--p 00000000 08:04 6587284 libhelloworld.so
b7faf000-b7fb0000 rw-p 00001000 08:04 6587284 libhelloworld.so
```

Zur Hausaufgabe 1.6: Bemerkungen

```
cat /proc/25419/maps
```

```
...
```

```
b7fd9000-b7fda000 r-xp 00000000 08:04 6587284 libhelloworld.so
b7fda000-b7fdb000 r--p 00000000 08:04 6587284 libhelloworld.so
b7fdb000-b7fdc000 rw-p 00001000 08:04 6587284 libhelloworld.so
```

Wie man sieht, wurde das File `libhelloworld.so` in den Adressbereich der beiden `main4b` Prozesse "gemappt". (einmal die Code-Sektion (`r-xp`), einmal die Daten-Sektion (`r--p`) und einmal als Copy-on-Write (`rw-p`))

Der Aufruf von

```
LD_LIBRARY_PATH=. strace ./main4b
```

gibt (für die Code-Sektion) Aufschluss darüber, wie dies passiert ist:

```
...
```

```
open("./libhelloworld.so", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0@\3\0\000"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=6314, ...}) = 0
mmap2(NULL, 8212, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7ad000
```

Zur Hausaufgabe 1.6: Bemerkungen

D.h. beim Aufruf von `main4b` wird (normalerweise durch `/lib/ld-linux.so.2`) zuerst das File `libhelloworld.so` mittels `mmap` read-only in einem Speicherbereich "virtualisiert". Wichtig ist hierbei desweiteren auch die File ID im `maps` Output:

```
b7fdb000-b7fdc000 rw-p 00001000 08:04 6587284 libhelloworld.so
```

Die Zahl `6587284` ist hierbei **nicht** die Dateigrösse, sondern eine eindeutige File-Kennung (Inode) die man mittels `stat libhelloworld.so` abfragen kann:

```
File: `libhelloworld.so'
Size: 6314  Blocks: 16  IO Block: 4096 regular file
Device: 804h/2052d  Inode: 6587289  Links: 1
...
```

D.h. obwohl wir zwei verschiedene Prozesse für `main4b` und auch in beiden Prozessen verschiedene Adressbereiche haben, benutzen doch beide dasselbe physikalische Library-File (und für die Bereiche des Files, die in den Speicher gemappt wurden, demzufolge auch denselben physikalischen Speicher). Im Gegensatz zu den statisch gelinkten Versionen, wo die Funktion `say_hello` mehrfach im (physikalischen) Speicher vorhanden ist.