Opdracht Besturingssystemen

4 mei 2015

Het doel van deze opdracht is de implementatie van een light-weight userspace threading bibliotheek. De opdracht kan worden opgedeeld in een aantal delen:

- Het implementeren van een aantal basis functies: creëeren van threads, context switching en opruimen van threads
- Het implementeren van het round-robin scheduling algoritme

Dit soort threads geven dezelfde gebruiksvriendelijkheid als OS threads maar geven enkele voordelen:

- Het is niet nodig om naar kernel space te gaan bij het switchen van user-space threads. Dit geeft iets minder CPU overhead.
- Synchronisatie is veel makkelijker aangezien de uitvoering maar binnen één thread gebeurt.

Het nadeel is uiteraard dat de uitvoering niet multi-threaded is en er geen winst is van meerdere cores. Het wisselen tussen de verschillende user-space threads (*Context switching*) dient te gebeuren op vrijwillige basis: wanneer een thread klaar is met zijn huidige werk en moet wachten op verdere input, zal hij dit laten weten aan de bibliotheek. Deze mag dan wisselen naar een andere thread.

1 Context switching

Context switches kunnen gerealiseerd worden aan de hand van de *setcontext()*, *getcontext()*, *makecontext()* en *swapcontext()* functies. *getcontext()* slaat de huidige register waarden en geblokkeerde signalen op in een buffer en *setcontext()* laad the registers en geblokkeerde signalen met de waarden die opgeslaan zijn in de buffer.

Listing 1: setcontext/getcontext voorbeeld

```
1 #include <ucontext.h>
2
3 volatile int finished = 0;
4
5 int main(int argc, char *argv[])
6 {
```

```
7       ucontext_t c;
8
9       printf("getcontext\n");
10      getcontext(&c);
11      printf("after getcontext\n");
12      if (!finished)
13      {
14          printf("if not finished\n");
15          finished = 1;
16          printf("setcontext\n");
17          setcontext(&c);
18          printf("after setcontext\n");
19      } else
20          printf("if finished\n");
21      printf("end of main\n");
22  }
```

Na uitvoer krijg je dit:

```
getcontext
after getcontext
if not finished
setcontext
after getcontext
if finished
end of main
```

Dus, bij een oproep van getcontext return je meermaals uit de functie. Je moet zelf bijhouden welk van de keren het is. De eerste keer is het net zoals een normale functie oproep. De andere keren zijn misschien onintuitief omdat dit tegen de normale programflow ingaat. Dan kom je namelijk uit deze functie terug na een oproep van setcontext.

Listing 2: makecontext/swapcontext voorbeeld

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <ucontext.h>
4
5   volatile int finished = 0;
6   ucontext_t c;
7   ucontext_t mainc;
8
9   void foo(void)
10  {
11      printf("foo\n");
12      printf("swapcontext\n");
13      swapcontext(&c, &mainc);
14      printf("after swapcontext\n");
15  }
16
17  int main(int argc, char *argv[])
18  {
19      printf("getcontext\n");
20      getcontext(&c);
21      printf("after getcontext\n");
22
23      printf("calloc stack\n");
24      char *stack = calloc(SIGSTKSZ, 1);
```

```
25      c.uc_stack.ss_sp = stack;
26      c.uc_stack.ss_size = SIGSTKSZ;
27      c.uc_link = NULL;
28
29      printf("makecontext\n");
30      makecontext(&c, foo, 0);
31      printf("after makecontext\n");
32
33      printf("getcontext main\n");
34      getcontext(&mainc);
35      printf("after getcontext main\n");
36
37      if (!finished)
38      {
39          printf("if not finished\n");
40          finished = 1;
41          printf("setcontext\n");
42          setcontext(&c);
43          printf("after setcontext\n");
44      } else
45          printf("if finished\n");
46      printf("end of main\n");
47  }
```

Na uitvoer krijg je dit:

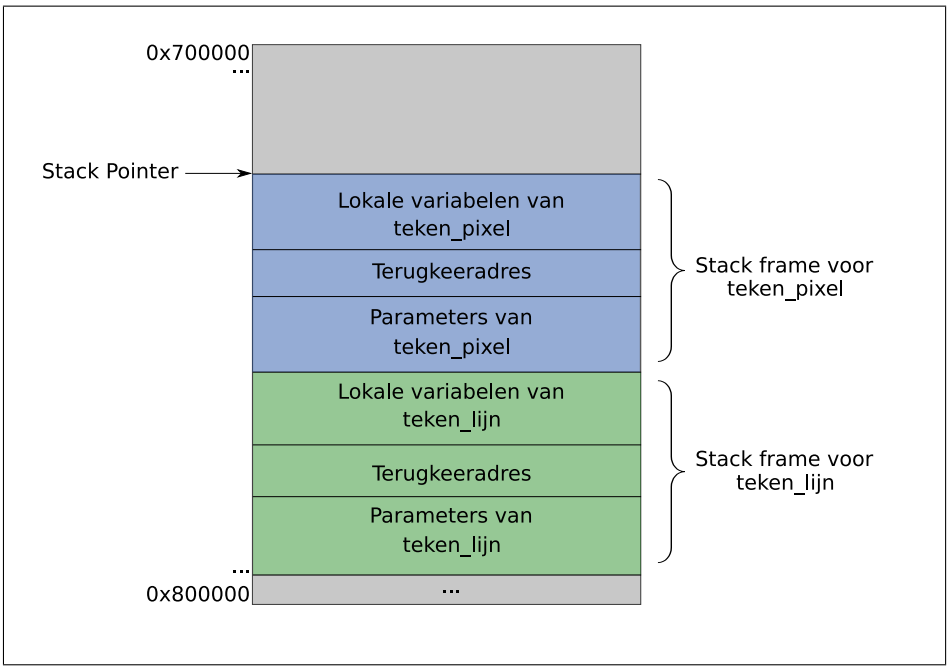
```
getcontext
after getcontext
calloc stack
makecontext
after makecontext
getcontext main
after getcontext main
if not finished
setcontext
foo
swapcontext
after getcontext main
if finished
end of main
```

Makecontext zorgt voor het aanmaken van een nieuwe context, die automatisch een opgegeven functie uitvoert. De swapcontext functie wisselt tussen twee contexten. Bij het implementeren van deze opdracht raden we aan enkel gebruik te maken van de drie functies: makecontext, getcontext en swapcontext. Dus, geen setcontext.

1.1 Context: Hoe werkt een callstack

Bij het oproepen van een functie worden de parameters van de functie, het terugkeeradres en de vorige stackpointer op een stack opgeslaan.

De stack wordt gebruikt voor het opslaan van lokale variabelen en bij het aanroepen van functies het terugkeeradres en de parameters.



Als je iets op de stack pusht, komt dit onderaan bij.

Bij het verlaten van een functie wordt de stack terug hersteld. De callstack is dan identiek aan de callstack voor de functie aanroep.

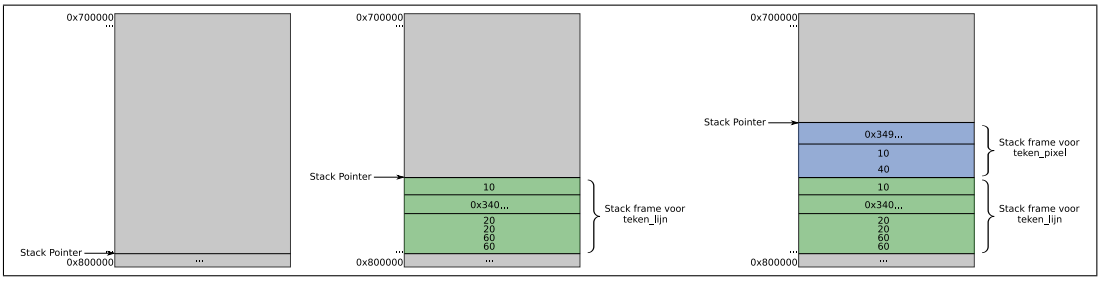
De huidige positie van de callstack wordt bijgehouden in de stackpointer en huidige positie in het programma wordt bijgehouden in de programcounter.

Bij het terugkeren uit een functie wordt de programcounter dan teruggezet op het opgeslagen terugkeeradres. Zodoende wordt de uitvoering dan verdergezet waar ze voor de oproep eindigde.

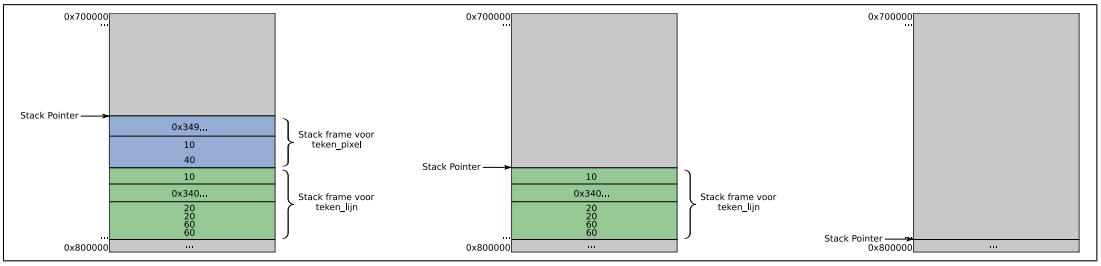
Listing 3: setjmp/longjmp voorbeeld

```
1  void teken_pixel(int x, int y)
2  {
3  }
4
5  void teken_lijn(int x1, int y1, int x2, int y2)
6  {
7      int i = 10;
8      teken_pixel(i,40);
9  }
10
11 int main()
12 {
13     teken_lijn(20, 20, 60, 60);
14 }
```

Onderstaande figuur toont de opbouw van de callstack. De linkertekening toont de callstack voor het oproepen van teken_lijn. De rechtertekening toont de callstack als teken_pixel in uitvoering is.



Onderstaande figuur toont de afbraak van de callstack. De linkertekening toont de callstack als teken_pixel in uitvoering is, de tekening in het midden illustreert hoe de callstack er uit ziet na het terugkeren uit teken_pixel naar teken_lijn. En de rechtertekening toont de callstack als we terug gekeerd zijn uit teken_lijn en de uitvoering van main hervatten.



1.2 Meerdere stacks

Elke thread moet voorzien zijn van een eigen stack. Anders gaat bij het switchen tussen threads, informatie op de callstack verloren gaan omdat de stackpointers ook worden terug gezet bij het switchen.

Volgende blok code implementeert dit:

```
thread->stack = (unsigned char *)calloc( SIGSTKSZ, 1 );
  thread->context.uc_stack.ss_sp = thread->stack;
  thread->context.uc_stack.ss_size = SIGSTKSZ;
  thread->context.uc_link = NULL;
```

2 Tips

Het is aangeraden om de verschillende delen incrementeel te implementeren. Om te beginnen is het nuttig om bekend te geraken met setcontext, getcontext, makecontext en swapcontext: deze functies vormen de basis van het context switchen.

3 Extra

3.1 I/O

Voeg operaties voor I/O toe aan de bibliotheek: usopen, usread, uswrite, usclose. Deze operaties zullen hetzelfde doen als de standaard calls maar op het opgeblik dat zo'n oproep zou blokkeren, zal de bibliotheek switchen naar een andere thread en later opnieuw proberen.

Om te detecteren en verhinderen dat een read of write blokkeert, dient er gebruik te worden gemaakt van non-blocking I/O. Het volgende stukje code zal een bestaande file-descriptor (fd) non-blocking maken en dient na de *open* oproep te gebeuren.

```
int setNonblocking(int fd) {
    int flags;

    if (-1 == (flags = fcntl(fd, F_GETFL, 0)))
        flags = 0;
    return fcntl(fd, F_SETFL, flags | O_NONBLOCK);
}
```

3.2 Programma

Schrijf een eenvoudig programma dat gebruik maakt van de functionaliteit van de bibliotheek. Bijvoorbeeld een producer thread die items uit een bestand leest en deze via een queue aan meerdere consumers geeft.