Kapitel 1

Worum geht es?

Es soll beschrieben werden, was die einzelnen Begriffe bedeuten, und wo derartige Konzepte eingesetzt werden.

1.1 Concurrency

Concurrency bedeutet, dass etwas gleichzeitig passiert. Dies kann auf verschiedenen Ebenen passieren. Aufgabe eines Betriebssytems ist es, dies zu verwalten. Früher war dies nur den Betriebssystemen vorbehalten. Durch das Aufkommen von Netzwerken und Multiprocessor Rechnern kommt es heutzutage häufiger vor, dass sich auch ein normaler Programmierer um diese Dinge kümmern muß.

1.2 Multiprogramming, Multitasking

Ein Betriebssystem hat das Problem verschiedene devices managen zu müssen. (Tastatur, Diskettenlaufwerk, Festplatte, CPU, Hauptspeicher, ..) Diese devices haben speziell bezüglich der Zugriffsgeschwindigkeit sehr unterschiedliche Charakteristika.

Prozessor	1ns (1GHz)	1 Sekunde
Cache	30ns	30 Sekunden
Hauptspeicher	200ns	200 Sekunden
Kontext Switch	10000ns	~1.2 Tage
Festplatte	10000000ns	110 Tage
Quantum	100000000ns	~3 Jahre

In dieser Tabelle werden mehrere Probleme deutlich. Erstens kann in der aktuellen Entwicklung die Festplatte nicht mit der Geschwindigkeitsentwicklung der CPU Schritt halten. Man benötigt sehr geschickte Methoden mittels Cache diese Lücke klein zu halten. Mit Kontext Switch ist die Zeit gemeint, die ein Betriebssystem braucht um zwischen zwei Prozessen zu wechseln. Ein Prozess ist dabei ein Programm, zusammen mit Information für das Betriebssystem, welches abgearbeitet werden soll. Was genau damit gemeint ist wird im zweiten Kapitel beschrieben. Mit Quantum ist die Zeit gemeint, die das Betriebssystem einem Prozess zur Verfügung stellt, bis der nächste dran kommt.

1.2.1 Multiprogramming

Das Verhältnis zwischen CPU und Festplatte zeigt, dass ein Prozess, der disk I/O betreibt nicht sehr effizient arbeitet. Bis was von der Platte zurückkommt könnte die CPU noch viele andere Dinge tun. Dazu verwendet man multi programming, dies bedeutet, dass zu einem beliebigen Zeitpunkt mehrere Prozesse auf die Abarbeitung warten. Um also besser zu werden führt das Programm einen Kontext Switch durch und lässt einen anderen Prozess laufen, während der ursprüngliche Prozess auf die Festplatte wartet.

1.2.2 Timesharing, Multitasking

Hier wird der Kontext Switch nicht nur durchgeführt, wenn er sich anbietet, z.B. beim Warten eines Prozesses, sondern auf regulärer Basis. Dies ist ein Merkmal des UNIX Betriebssystems. Um dies zu Implementieren wird ein sog. Timer verwendet, ist dieser abgelaufen wird der nächste Prozess abgearbeitet. Effizient ist ein solches System wenn die Zeit für den Kontext Switch klein ist im Vergleich zu der Zeit, die ein Prozess in der CPU bleibt. Dies ist das Verhältnis Quantum/Kontext Switch. An dieser Stelle sieht man auch den Vorteil eines Multi-Prozessor Systems, da können mehrere CPUs die anstehenden Prozesse erledigen.

1.3 Wie wird Concurrency implementiert

Wenn verschiedene Dinge gleichzeitig passieren, braucht man Mittel zur Kommunikation. Man unterscheidet zwischen Hardware und Softwarelösungen.

1.3.1 Interrupts

Dies sind Hardware Signale, die dem Prozessor mitteilen, dass irgendetwas passiert ist. Wenn der Prozessor merkt (flag) dass ein Interrupt ausgelöst wurde unterbricht er die normale Arbeit (speichert den aktuellen Stand) und macht mit der Interrupt Service Routine weiter. Man unterscheidet hier zwischen asynchronen events (unabhängig vom aktuellen Befehl in der CPU) und synchronen events, (z.B. Division durch 0), die nur bei speziellen CPU Befehlen auftauchen können. Ein typisches Beispiel eines asynchronen events sind Interrupt ausgelöst durch externe devices (z.B. Netzwerk-Karte). Die device Treiber sind typische interrupt service routines, sie werden aufgerufen wenn z.B. von der Festplatte Daten kommen. Interrupts werden auch verwendet um das timesharing zu implementieren, nach einer vorgegebenen Zeit (quantum) wird ein interrupt ausgelöst (asynchron) und der nächste Prozess wird gestartet.

1.3.2 Signal

Dies passiert auf Softwareebene. Das bedeute, dass diese auch per Software ausgelöst werden können. Ähnlich wie mit der CPU und den interrupts, gehören zu einem Process (UNIX) mehrere flags, die gesetzt werden wenn ein Signal eintrifft. Wieder kann man zwischen synchronen und asynchronen Signalen unterscheiden. Auch entsprechen viele Signale interrupts auf Hardwareebene (z.B divide by zero). Die Signale werden vom Betriebssystem oder direkt von interrupt service routinen an die jeweiligen Prozessse gesendet. Dieses Signal kann von den jeweiligen Prozessen abgefangen werden (dann existiert ein signal handler) oder ignoriert werden, oder das entsprechende Signal kann nicht abgefangen werden, dann wird der entsprechende Prozess beendet. Signal handler müssen in spezieller Weise geschrieben werden, da es passieren kann, dass diese Routine mehrfach und gleichzeitig (concurrency) aufgerufen wird.

1.3.3 I/O

Wie schon angedeutet ist es für ein gutes Betriebssystem wichtig devices mit unterschiedlichen charakteristischen Zeiten effektiv zu handhaben. Eine Methode ist Multiprogramming. Es gibt aber auch Methoden, damit ein einzelner Prozess nicht durch das Warten blockiert wird. Die Methoden sind asynchronous I/O und spezielle threads für das I/O. Ähnliches passiert auch bei Netzwerkprogrammen. Dort gibt es ähnliche Methoden, die in einem späteren Kapitel beschrieben werden. (polling, threads, select, asynchronous I/O).

1.3.4 thread

Damit man concurrency zwischen Prozessen erreicht, gibt es in UNIX die Aufrufe fork. wait und exec. Dies wird im zweiten Kapitel besprochen. Wie diese parallelen Prozesse kommunizieren wird im weiteren noch intensiv besprochen. (pipes, signals, FIFO, semaphore, shared memory, messages). Ein anderes Thema ist concurrency innerhalb eines Prozesses, dann spricht man von threads. Ein thread ist normalerweise die Kette von CPU Befehlen die zu einem Prozess gehören. Man kann nun versuchen auf einem Mehrprozessor System schneller werden, indem man das Programm per Softwarebefehle in threads zerlegt, die gleichzeitig ausgeführt werden. Oder aber wenn z.B. I/O mit langsamen devices in ein eigenes thread gepackt wird.

1.3.5 network

Hier tritt concurrency in Form von Prozessen auf verteilten Rechnern auf. Das sog. client/server Modell wird verwendet um Kommunikation zwischen verteilten Systemen zu ermöglichen. Ein client fragt nach einer Leistung nach (z.B. Zugriff auf eine Datenbank), der Server gewährt sie. Auf Programmierebene wird dies durch remote procedure calls (RPC) ermöglicht. Eine fortschrittlichere Methode ist das Konzept eines objekt orientierten Netzwerk modells. Die Rechner(server) sind Objekte mit definierten Message systemen. Ein weiteres Konzept ist das zur Verfügung stellen eines parallen Interfaces und drunter liegt ein Netzwerk oder ein paralleler Rechner (PVM).

1.4 UNIX Standard

Ein Problem bei UNIX ist die Standardisierung. Wichtige sind hier ANSI, POSIX und Spec 1170. Durch ANSI wurde die Programmiersprache C standardisiert. Durch IEEE wurde mittels POSIX ein Standard auf Betriebssystem Ebene geschafen. D.h. hält sich ein Programm an die POSIX Systemaufrufe, so sollte es portabel sein. Nicht nur zwischen UNIX Varianten, sondern zwischen allen Systemen die ein POSIX API bieten. Speziell dies war auch ein Designkriterium des POSIX interfaces. System V Release 4 und Spec 1170 (von X/Open) sind weitere weitverbreitete Standards. Die Beispiel aus R&R halten sich an POSIX und wenn dies nicht möglich (da nicht standardisiert) ist an Spec 1170.

1.5 UNIX Programmierung

Natürlich werden im weiteren verschiedene Beispiele besprochen. Diese werden als UNIX C Programme vorgestellt. Daher einige Bemerkungen zur Verwendung von C Programmen unter UNIX. UNIX stellt eine Reihe von sog. Systemroutinen zur Verfügung und außerdem sog. library Funktionen. Der Unterschied ist, dass die System Funktionen direkt im Kernel mode laufen, d.h. vollen Zugriff auf sämtliche Betriebssystem Daten haben. Die library Funktion (z.B. printf) sind lediglich hilfreiche Routinen wenn man eigne Programme schreibt (z.B. printf) machmal verwenden diese Systemaufruf manachmal aber auch nicht. Systemfunktionen werden im Abschnitt 2 des UNIX manual beschreiben, Library funktion in Abschnitt 3. Verwendet man diese Funktion (was man stets tun sollte) so ist es nüztlich vorher den entsprechenden Abschnitt des manuals zu studieren. Dazu ein Beispiel mit dem Systemaufruf open, die man page gibt folgende Information:

```
open(2)
NAME
  open, creat - Open a file for reading or
writing
SYNOPSIS
  #include <fcntl.h>
  #include <sys/stat.h>
  #include <sys/types.h>
  int open (
            const char *path,
            int oflag [ ,
            mode_t mode ] );
```

Man sieht welche Header Dateien eingebunden werden müssen. Man sieht die Synopsis, d.h. wie die Funktion aufgerufen wird. Im restlichen Teil sieht man die Bedeutung der Parameter. Auch kommt ein Teil :

```
Interfaces documented on this reference page
conform to industry standards

as follows:

open(): POSIX.1, XPG4, XPG4-UNIX
```

d.h. man erfährt, dass die Funktion open dem POSIX Standard genügt.

1.5.1 Fehler innerhalb von System Routinen

Eine wichtige Frage ist, was passiert in einer Fehlersituation. Die Routine open versucht einen file zu öffnen. Nun kann sein, die Datei ist nicht da (beim Lesen) oder aber irgendwelche Zugriffsrechte passen nicht, oder aber es gibt nicht mehr genug filedescriptoren. Die Methode bei UNIX Routinen ist, dass der Rückgabewert auf -1 oder NULL gesetzt wird und die Systemvariable errno sagt, welcher Fehler vorlag. Man muß aufpassen wenn man die Variable errno liest, da diese nur bei einem neuen Fehler geschrieben wird. Liegt ein Fehler vor kann man mit dem Befehl perror() sich einen entsprechenden Text ausgeben:

```
SYNOPSIS

  #include <stdio.h>
  void perror( const char *string);
```

und bei der Verwendung wird die Erklärung des Felers an den übergebenen Text angehängt. Dazu ein Ausschnitt:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
 int fd;
 if ((fd = open("my.file", O_RDONLY)) == -1)
     perror("Unsuccessful open of my.file");
```

und falls die Datei my.file nicht zum Lesen geöffnet werden kann (da nicht vorhanden) erhält man den folgenden Fehler:

```
Unsuccessful open of my.file: No such file or directory
```

Es kann viele verschiedene Gründe geben, sie werden alle auf der man page aufgelistet. Ein Grund kann sein EAGAIN, d.h. der file ist gelockt, d.h. er wird momentan von einem Process bearbeitet und kann nicht gelesen werden. Dann muss man probieren bis er wieder frei ist

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
  int fd;
  while (((fd = open("my.file", O_RDONLY)) ==
-1)
                              && (errno ==
EAGAIN));
  if (fd == -1)
        perror("Unsuccessful open of my.file");
```

Man beachte die geschickte Verknüpfung mittels &&. Eine andere Möglichkeit einen Fehlerwert als Text auszugeben ist hier:

```
#include <string.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
  int fd;

  if ((fd = open(argv[1], O_RDONLY)) == -1)
    fprintf(stderr, "Could not open file %s:
%s\n",
                argv[1], strerror(errno));
```

1.5.2 gute C Routinen

Die C Systemroutinen sind meist ein gutes Beispiel für Tips und Tricks beim Programmieren. Hier einige Dinge die man beherzigen sollte.

- Kommunizieren mittels return.
- kein exit innerhalb von Routinen.
- keine impliziten Annahmen über die Grösse von buffern und arrays.
- wenn Limits gesetzt werden müssen, dann verwende System Variablen und keine eigenen Konstanten.

- Verwende die Standard library sooft wie möglich.
- Verändere nicht die input Parameter. (geht sowieso nur bei pointern)
- Bei malloc aufpassen, dass der Speicher wieder freigegeben wird.
- Verwende sooft wie möglich automatische Variablen, keine static Variablen und keinen dynamischen Speicher.
- Wie ist es bei rekursiven Aufruf, wie bei threads?

1.5.3 strtok

Als Beispiel zu diesen Kriterien betrachte folgendes Problem: Beim Aufruf eines C-Programms wird die Kommandozeile als array von strings übergeben.

```
int main (int argc, char *argv[])
```

Irgendwo im Betriebssystem muss dieser Datentyp aufbereitet werden, denn beim Aufruf des Programms in der shell hat man nur einen string zur Verfügung. Dies soll mit der Funktion makeargv gemacht werden, die wir schreiben wollen. Eine Beispielanwendung:

```
#include <stdio.h>
int i;
char **myargv;
char mytest[] = "This is a test";
int numtokens;
if ((numtokens = makeargv(mytest, &myargv)) <
0)
  fprintf(stderr, "Could not construct an
argument array\n");
else
  for (i = 0; i < numtokens; i++)
            printf("%i: %s\n", i, myargv[i]);
```

Die Funktion makeargv ist vom Typ int und hat als letzten Parameter das Ergebnis mittels Übergabe bei Referenz, d.h. einen Zeiger mehr. Die Alternative wäre als Rückgabewert der Funktion, und NULL im Fehlerfall. In der UNIX Umgebung genügt dies noch nicht, da als Trennsymbole nicht nur blanks, sondern sog. whitespaces erlaubt sind, so z.B. auch Tabs. Deshalb hier ein Programm, wie wir es wirklich haben wollen:

```
#include <stdio.h>
#include <stdlib.h>
int makeargv(char *s, char *delimiters, char ***argvp);
void main(int argc, char *argv[])
  {
  char **myargv;
  char delim[] = " \t";
  int i;
  int numtokens;
  if (argc != 2) {
     fprintf(stderr, "Usage: %s string\n", argv[0]);
     exit(1);
  }
  if ((numtokens = makeargv(argv[1], delim, &myargv)) <
0) {
      fprintf(stderr,
            "Could not construct argument array for
%s\n", argv[1]);
      exit(1);
      }
  else {
      printf("The argument array contains:\n");
      for (i = 0; i < numtokens; i++)
          printf("[%d]:%s\n", i, myargv[i]);
      }
  exit(0);
  }
```

Ein derartiges makeargv wollen wir implementieren. Dazu verwenden wir die Systembibliothek Funktion strtok. Diese Funktion spaltet einen String in mehrere Tokens auf. Die man page sagt folgendes:

```
strtok(3)                                        strtok(3)
strtok - Split string into tokens
SYNOPSIS
#include <string.h>
 char *strtok(
        char *s1,
       const char *s2);
...
DESCRIPTION

The strtok() function splits the string pointed to
by the s1 parameter into a sequence of tokens,
each of which is delimited by a byte equal to one
of the bytes in the s2 parameter.
Usually, the strtok() function is called repeatedly
to extract the tokens in a string.
The first time the application program calls the
strtok() function, it sets the s1 parameter to
point to the input string.
The function returns a pointer to the first token.
Then the application program calls the function
again with the s1 parameter set to the null
pointer.
This call returns a pointer to the next token in
the string.
The application program repeats the call to
strtok()
with the s1 parameter set to the null pointer until
all the tokens in
the string have been returned.
```

Um eine Funktion makeargv zu schreiben geht man wie folgt vor:

- Erzeuge eine Kopie des strings, der zerlegt werden soll.
- Durchlaufe diesen mit strtok um die Anzahl der tokens zu bestimmen.
- erzeuge ein entsprechend grosses array.
- Durchlaufe den string nochmal mit strtok und speichere die tokens.

Das Ergebnis ist folgende Funktion:

```
#include <string.h>
#include <stdlib.h>
/*
 * Make argv array (*arvp) for tokens in s which are
separated by
 * delimiters. Return -1 on error or the number of tokens
otherwise.
*/
int makeargv(char *s, char *delimiters, char ***argvp)
{
 char *t,*snew;
 int numtokens,i;

/* snew is real start of string after
   skipping leading delimiters */

 snew = s + strspn(s, delimiters);
/* create space for a copy of snew in t */

 if ((t = calloc(strlen(snew) + 1, sizeof(char)))==NULL)
    {*argvp = NULL;numtokens = -1;}
 else { /* count the number of tokens in snew */
    strcpy(t, snew);
    if (strtok(t, delimiters) == NULL) numtokens = 0;
    else for (numtokens = 1;
           strtok(NULL, delimiters) != NULL;
           numtokens++);
/* create an argument array to contain ptrs to tokens */
    if ((*argvp=calloc(numtokens+1, sizeof(char *)))==NULL)
      {free(t);numtokens = -1;}
    else
      { /* insert pointers to tokens into the array */
      if (numtokens > 0) {
        strcpy(t, snew);
        **argvp = strtok(t, delimiters);
        for (i = 1; i < numtokens + 1; i++)
           *((*argvp) + i) = strtok(NULL, delimiters);
        }
      else {**argvp = NULL;free(t);}
      }
    }
return numtokens;
}
```

1.6 reentrant-safe

Die Funktion strtok ist ein schlechtes Beispiel einer System funktion. Man sollte dieses Design nicht kopieren. Die Funktion hat eine lokale statische Variable, um sich den Platz innerhalb des Strings, der zerlegt wird zu merken. Dies kann Probleme verursachen, wenn diese Funktion z.B. innerhalb eines signal-handlers verwendet wird, und auch innerhalb des Hauptprogramms. Das Signal kommt während der Abarbeitung im Hauptprogramm. Dann wird die Arbeit im Hauptprogramm unterbrochen und der signal-handler nimmt die Arbeit auf. Dies führt zu einem Fehler bei der Rückkehr in das Hauptprogramm. Das gleiche Problem tritt auf, wenn man ein Programm in threads zerlegt, und sich mehrere threads in der gleichen Funktion aufhalten. Funktionen, die dieses Problem nicht haben, heißen reentrant-safe. Man kann noch genauer unterscheiden, zwischen async-signal-safe und thread-safe. Es ist schwieriger aysnc safe zu erreichen. Unter POSIX gibt es für fast alle Funktionen eine thread-safe Variante, sie hat den gleichen Namen mit der Extension _r.

```
strtok(3)                                    strtok(3)
NAME

strtok, strtok_r - Split string into tokens

SYNOPSIS

 char *strtok_r(
             char *s1,
             const char *s2,
             char **savept);
```

und der letzte Parameter dient zum Speichern der Position im string.

1.6.1 error handling und thread-safe

Ein anderes Problem ist das error-handling der System Funktionen. Dies geschieht über eine globale Variable errno. Dies gibt Probleme bei der Verwendung von mehreren threads. Als Lösung wurde in POSIX definert, dass die neuen Funktionen unterschiedliche Fehlercodes zurück liefern. Bei den schon vorhanden Funktionen kann man unterschiedliche Ansätze wählen um diese reentrant-safe zu machen. Als Beispiel betrachten wir die Funktion read.

```
SYNOPSIS

#include <unistd.h>

ssize_t read(
       int filedes,
       void *buffer,
       size_t nbytes);
```

- Verwende stattdessen int read_r(int filedes, void *buffer, size_t nbytes, ssize_t *bytesread); und man hat den Rückgabewert frei zur Fehlerbehandlung.
- Verwende ssize_t read_r(int filedes, void *buffer, size_t nbytes, int *status); und man kann die Fehlerbehandlung im letzten Parameter verstecken.
- Man macht die Variable errno zur lokalen Variable in den threads.
- errno wird ein Macro, welches eine Funktion aufruft, die per thread arbeitet.
- Implementiere die Fehlerbehandlung von read als software signal.

Alle Methoden haben Nachteile: In den beiden ersten Fällen müssen die vorhandenen Programme geändert werden. Die dritte Methode benötigt eine spezielle Unterstützung durch Compiler und Linker. Bei der vierten Lösung müssen die Funktionen die errno setzen entsprechend geändert werden. Die fünfte Lösung benötigt einen speziellen neuen signal-handler. Außerdem ist drei und vier nicht async-safe.

1.7 Aufgabe

Teste die Funktion makeargv mit dem vorgestellten Hauptprogramm. Dazu soll die Funktion makeargv in einer eigenen Datei argvlib.c sein. Verwende ein makefile um aus den beiden Dateien eine ausführbare Datei zu machen. Teste mit unterschiedlich inputs. Teste dabei die Randbedingungen.