Lyubomira Dimitrova, Dung Nguyen, Maryna Charniuk

Aufgabe 1

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <sys/wait.h>


int main()
{
        // 2 pipes
        // first pipe to send input string from parent to child process
        // second pipe to send reversed string from child to parent process
        int fd1[2]; // first pipe stores the reading and writing end
        int fd2[2]; // second pipe stores the reading and writing end

        // default input
        char input[] = "Hello World";

        // char input[100];
        //scanf("%s",input);

        int pid;

        if (pipe(fd1) ==-1){
                fprintf(stderr, "first Pipe Failed");
                return 1;
        }

        if (pipe(fd2)==-1){
                fprintf(stderr, "second Pipe Failed");
                return 1;
        }

        // create child process B of parent process A
        pid = fork();

        if (pid < 0){
                fprintf(stderr, "fork failed");
                return 1;
        }
        //parent
        else if (pid > 0){
                char output[strlen(input)+1];
```

```
                close(fd1[0]); // Close reading ends of pipe 1

                //Write input string and close writing end of pipe 1
                write(fd1[1], input, strlen(input)+1);
                close(fd1[1]);

                // Wait for child to send a string
                wait(NULL);

                close(fd2[1]); //Close writing end of pipe 2

                // Read string from child, print and close reading end of pipe 2
                read(fd2[0], output, strlen(input)+1);
                printf("output: %s\n", output);
                close(fd2[0]);
        }

        // child
        else if (pid == 0){
                close(fd1[1]); // Close writing ends of pipe 1

                // Read input string using pipe 1
                char output[strlen(input)+1];
                read(fd1[0], input, strlen(input)+1);

                // reverse the string
                int i;
                int k = 0;

                for (i=strlen(input)-1; i>=0; i--){
                        output[k++] = input[i];
        }

                // Close both reading ends
                close(fd1[0]);
                close(fd2[0]);

                // Write output in pipe 2 and close writing end of pipe 2
                write(fd2[1], output, strlen(output)+1);
                close(fd2[1]);
                exit(0);
        }
}
```

Aufgabe 2

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <stdlib.h>
int main(){
    int i, shmID, *shared_mem, count=0, total=0,rnd;
    shmID = shmget(IPC_PRIVATE, sizeof(int), IPC_CREAT | 0644);
    shared_mem = (int*)shmat(shmID,0,0);
    *shared_mem = 0;
    if (fork())
        for (i=0; i<500; i++){
            *shared_mem+=1;
            printf("\n Elternprozess: %i", *shared_mem);
            sleep(2);
        }
    else
        for (i=0; i<500;i++){
            *shared_mem+=1;
            printf("\n Kindprozess: %i", *shared_mem);
            rnd=rand();
            sleep(rnd%3);
        }
    shmdt(shared_mem);
    shmctl(shmID, IPC_RMID, 0);
    return 0;
}
```

- **int shmget(key_t key, size_t size, int shmflg)** - Shared Memory Segment (SMS) wird allokiert. Rückgabewert: ID des neuen Segments.
    * **key**, z.B. **IPC_PRIVATE** - numerischer Schlüssel, der dem SMS zugewiesen wird.
    * **size**, z.B. **sizeof(int)** - Größe des SMS.
    * **shmflg**, z.B. **IPC_CREAT | 0644** - enthält Zugriffberechtigungen, die in Oktalformat dargestellt sind, und Steuerbefehle.
- **void *shmat(int shmid, const void *shmaddr, int shmflg)** - angefordertes SMS wird in den Adressraum aufgenommen. Rückgabewert: Adresse des neuangehängten Segments.
    * **shmid** - die ID eines SMS (Rückgabewert von **shmget**)
    * **shmaddr**, z.B. **NULL** - wo im Adressraum das SMS angefügt werden soll. Wenn **NULL**, entscheidet das System.
    * **shmflg** - enthält andere Flags
- **int shmdt(const void *shmaddr);** - SMS wird getrennt, danach darf nicht mehr benutzen werden.

    * **shmaddr** - Zeiger auf SMS, der von shmat erstellt wird.

- **int shmctl(int shmid, int cmd, struct shmid_ds *buf);** - führt die Operation **cmd** auf dem SMS **shmid** aus. Z.B, um ein SMS zu löschen, **shmctl(shmID, IPC_RMID, 0)**.

In der gegebenen Funktion ist es schwierig ein genaueres Ergebnis vorherzusagen, da im Kindprozess eine Random-Methode `rand()` benutzt wird und man weißt nicht an welcher Stelle ein Wechsel zwischen Kind- und Elternprozess passiert. Z.B., die ersten zwei Zeilen der Ausgabe könnten dadurch erklärt werden: Der Elterprozess wird genau dann unterbrochen, wenn der Parameter `*shared_memory` der `printf` Funktion schon ausgewertet worden ist (der Wert 1 wurde ausgelesen), aber die printf Funktion noch nicht vollständig ausgeführt wurde.

```
Kindprozess: 2
Elternprozess: 1
Kindprozess: 3
Kindprozess: 5
Kindprozess: 6
Elternprozess: 4
...
```

Aufgabe 3

1. 256 Dateien. Der Befehl `ulimit -n` gibt die maximale Anzahl geöffneter Dateien per Prozess zurück. Es gibt zwei solche Limite: *hard* (-H) und *soft* (-S).

   Limits auf Fedora 29 (Kernel 5.0):

   - `ulimit -Hn`

     `4096`
   - `ulimit -[S]n`

     `1024`

   Das *soft* Limit kann von jedem Nutzer geändert werden (`ulimit -[S]n 4096`), aber nur bis zum Wert des *hard* Limits. Das *hard* Limit kann nur vom `root`-Nutzer geändert werden (`ulimit -Hn 8196`).

   Quellen: Stackexchange [1], `man ulimit`

2. Zeile 856: `struct files_struct *files;`

3. Link zum Quelltext[2], Zeile 916.

   Angenommen, dass jeder Eintrag 4 Bytes benötigt, wäre der minimale Speicherbedarf der `file` Datenstruktur ∼90 Byte.

[1] `https://unix.stackexchange.com/questions/36841/why-is-number-of-open-files-limited-in-linux`
[2] `https://elixir.bootlin.com/linux/latest/source/include/linux/fs.h#L916`

4. Nein, den Dateideskriptor finden wir nicht in der `file` Datenstruktur. Aus der Figur im Abschnitt 4.4 des Buches geht hervor, dass die `files_struct` Datenstuktur ein Array `fd[256]` enthält, dessen Elemente vom Typ `*file` sind. Die Indizes der Feldelemente sind im Endeffekt die Dateideskriptoren. Im Quellcode[3] heißt das Array `fd_array` (Zeile 66). Ein Grund für diese Implementierung könnte die einfachere Umleitung sein. Eine geöffnete Datei könnte durch mehreren Dateideskriptoren zugänglich sein, z.B. nach einem Aufruf von `dup2`. In dem Fall müsste die Datenstruktur der Datei (Typ `file`) selbst nicht geändert/dupliziert werden (wie es bei einer zusätzlichen Variable `fd` wäre), sondern einfach der Pointer dazu.

Aufgabe 4

Im eins-zu-eins Ausführungsmodell muss man bei jedem Threadwechsel sowohl den Kernel- als auch den Benutzerraum wechseln, was zusätzliche Zeit benötigt. Der Kernelscheduler weißt nichts davon, was genau im Benutzerraum gemacht wird. Aus diesem Grund trifft der Kernel nicht immer sinnvolle Entscheidungen für diesen Wechsel.

Das Overhead hängt davon ab, wo die Threads ausgeführt werden. Am schnellsten passiert der Threadwechsel, wenn nach dem Zufallsprinzip die Threads von dem gleichem CPU und an dem gleichen Socket ausgeführt werden. Etwas mehr Zeit braucht man, wenn die Threads sich an verschiedenen Sockets befinden und noch mehr Zeit, wenn sie von verschiedenen CPUs ausgeführt werden und an verschieden Sockets sich befinden.

Die Mehrheit der durch einen Threadwechsel entstehenden Kosten ergeben sich aber durch den Scheduler - die komplexe Entscheidung, was und wo auszuführen.

Aufgabe 5

1.
```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#define NUM_THREADS 500000

void *TaskCode(void *argument) {
    int tid;
    tid = *((int*) argument);
    printf("It's me, dude! I am number %d!\n", tid);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
```

[3] https://elixir.bootlin.com/linux/latest/source/include/linux/fdtable.h#L66

```
    int rc;
    int i;

    for(i = 0; i < NUM_THREADS; ++i) {
        thread_args[i] = i;
        printf("In main: creating thread %d\n", i);
        //create all threads:
        rc = pthread_create(&threads[i], NULL, TaskCode, (void *) &thread_args[i]);
        rc = pthread_join(threads[i], NULL); //wait for the thread to complete
        assert(0 == rc);
    }
    exit(EXIT_SUCCESS);
}
```

2. Mit `NUM_THREADS = 500 000` hat der Befehl `time ./threads` folgende Ausgabe:

```
In main: creating thread 0
It's me, dude! I am number 0!
In main: creating thread 1
...
It's me, dude! I am number 499998!
In main: creating thread 499999
It's me, dude! I am number 499999!

real    0m10.107s
user    0m1.280s
sys     0m7.296s
```