BUS Exercise 3
Group 23

Thilo Metzlaff	Mats Frenk	René van Emelen
406247	393702	406008

May 27, 2020

Contents

1 Linux Processes and Child creation **4**
- 1.1 Controlled Forking . . . 4
- 1.2 Streams . . . 5
- 1.3 Undead Processes . . . 5
- 1.4 Linking Children . . . 5
- 1.5 Parent/Child execution priority . . . 5

2 Tree Hugging **7**
- 2.1 The Family Tree . . . 7
- 2.2 Parent-Child execution order . . . 7
- 2.3 Wait for Order 66 . . . 7

3 Process data communism **9**
- 3.1 Stuffing things in Pipes or cooperative memory use? . . . 9
- 3.2 . . . 9
- 3.3 . . . 10

Introduction

Welcome back to this special endeavour, where i write every single thing in LaTeX; Please be patient, while we look for the missing full-stop and enjoy a semicolon instead.
Now... How was it that i defined all this again? Lemme look for the definition real quick...
Ah yes, there it is:

THE FORMAT

- Every file will be named similar to the sections in here, so `2.1-stack_exercise.c` is Exercise 2, section 1.
- Every Solution **WILL** be in this pdf, but not necessarily anything predefined by the exercise.
- Any explanation will be both in this PDF as well as in each file.
- This explanation will be in each PDF, in case someone who doesn't know the format tries to correct the exercises
- **WARNING:** Humor may or may not be used. If you are allergic to humor, that sounds like a personal problem.

1 Linux Processes and Child creation

1.1 Controlled Forking

To control the output of parent and child, we need another `if-else` construct, testing the return value of `fork()`. If said value is 0, we are in the child and we can execute `child_B_proc()`, otherwise we execute `child_A_proc()`. The final code looks like this:

```c
#include <stdio.h>
#include <unistd.h>

void child_B_proc()
{
  while (1)
  {
    fprintf(stdout, "%s", "C");
    fflush(stdout);
  }
}

void child_A_proc()
{

  while (1)
  {
    fprintf(stdout, "%s", "A");
    fflush(stdout);
  }
}

void parent_proc()
{
  while (1)
  {
    write(1, "B", 1);
  }
}

int main(void)
{
  // follow parent process & continue child A here
  if (fork() == 0)
  {
    // follow child A & continue child B here
    // essentially: spawn B from A and see where we are
    // then execute code accordingly
    if (fork() == 0)
      // execute code for child B
      child_B_proc();
    else
      // execute code for child A
      child_A_proc();
  }
  else
    parent_proc();
  return 0;
}
```

3.1_prozesse.c

1.2 Streams

`stdin` and `stdout` are so-called "streams". A stream is a temporary memory space that we can use to write to and communicate. This ensures that we only write to a specific location and only when data is complete. `stdin` and `stdout` are the program's IO streams. Reading from `stdin` will read any input that has been given to a program and reading `stdout` wil give its output. That is how the linux terminal reads the program's output as well.
The problem in a stream is the fact that they persist, unless we manually flush them. So writing `"aaaaaaaaa"` into `stdout` and then writing `"test"` to it, will result in the outputs `"aaaaaaaaa"` and `"testaaaaa"`, because we overwrite the stream without flushing it. That's why you should firs to flush any stream before writing to it.
printf(**char***, \dots) and fprintf(**char*** buf, **char***,...) don't flush, but rather they pad the output to the next byte and add a **NULL** terminator instead. To prevent this, we can use fflush(buf) and completely flush the stream. Both functions also parse the string to look for formatting and where to insert variables, before using write(**int** fd, **char*** buf, **int** blen); In this case, `write()` is more efficient, because `write()` doesn't check the string. It writes it to stdout.

1.3 Undead Processes

A Zombie process is simply said: a process with daddy issues.
Zombie processes are created, when the child's Process Descriptor isn't updated in the parent process and the child terminates unexpectedly. In this case, the prozess is dead, but its descriptor still exists, so the parent thinks the process is still running and leaves it registered. To prevent this, one can simply use the `wait()` and `exit()` syscalls, this makes the parent wait and update the Status of the child as soon as it commits suicide (terminates) with `exit()`;

1.4 Linking Children

A child process always has its parent's PID linked, so it can be identified by the system and organized appropriately. If this wasn't the case, the system couldn't create a proper process tree structure and would fail to execute a program.

1.5 Parent/Child execution priority

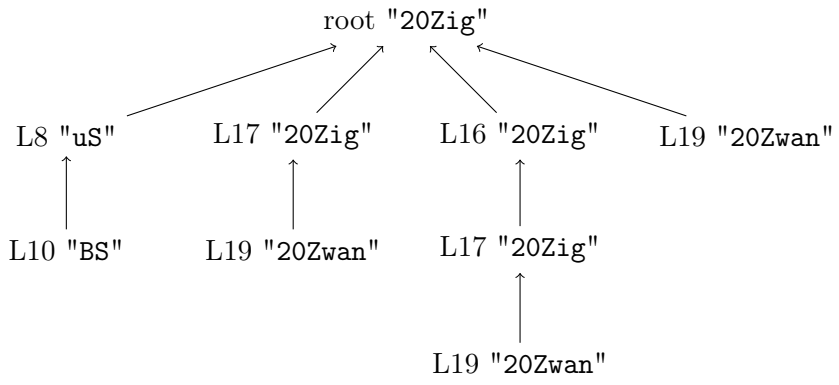
Using the program `charcount.c`, one can pipe `prozesse.c` into it and observe, that the percentage frequency is approximately 30%. This percentage drifts around, due to the random execution order. In long term observation, one can see that the frequency will drift further, suddenly prioritizing a single process A LOT stronger, but then changing between them. The conclusion would be, that, on average, all processes have the same priority and frequency.

```c
#include <stdio.h>
#include <unistd.h>
#define BUFFERLEN 6000
int main(void)
{
    char buffer[BUFFERLEN];
    int a = 0;
    int b = 0;
    int c = 0;

    double percentagea = 0;
    double percentageb = 0;
    double percentagec = 0;

    // main loop
    while (1)
    {
        sleep(10);
        a = 0;
        b = 0;
        c = 0;
        fgets(buffer, BUFFERLEN, stdin);
        for (int i = 0; i < BUFFERLEN; i++)
        {
            if (buffer[i] == 'A')
            {
                a++;
            }
            else if (buffer[i] == 'B')
            {
                b++;
            }
            else
            {
                c++;
            }
        }

        percentagea = ((double)a / (a + b + c)) * 100;
        percentageb = ((double)b / (a + b + c)) * 100;
        percentagec = ((double)c / (a + b + c)) * 100;
        printf("Occurence of A: %d, B: %d, C: %d. \n In percent: \n  A: \e[36m%.3f\e[0m %% \n  B: \e[36m%.3f\e[0m %% \n  C: \e[36m%.3f\e[0m %%\n", a, b, c, percentagea, percentageb, percentagec);
    }
}
```

3.1_charcounter.c

2 Tree Hugging

2.1 The Family Tree



2.2 Parent-Child execution order

The child process and the parent process have the same priority. Therefore the OS has to decide which process to execute first, by randomly chosing. This makes the order of execution random, unless the parent awaits the child process's execution.

2.3 Wait for Order 66

Since we want a deterministic program output, namely `"BuS 20ZwanZig"`, we have to wait for each child to do its thing. How does one wait for a child to terminate itself, you might ask? Simply by using **exit**(0); in the child process. To use wait and exit, we have to do **#include** <stdlib.h> and **#include** <sys/**wait**.h>. Now we are able to use **wait** and **exit**.
Since we can just use **wait**(**NULL**); to block a parent until all children are done, we just use that a bunch of times. Before we can get a fully deterministic output, we have to end some processes early, to only get the unique output. we can do this by checking if **fork**(); returned 0, meaning we are in a child, and then terminating early. I terminated all child processes before the 20, they are useless here, because they give duplicate output.

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

int main()
{
    int await1 = fork();
    if (await1 == 0){
        if (fork() == 0){
            printf("B \n");
            exit(0);
        }
        else
        {
            wait(NULL);
            printf("u \n");
        }
        printf("S \n");
        exit(0);
    }else{
        wait(&await1);
        int await3 = fork(); // makes a child
        int await4 = fork(); // makes 2 children

        if (await4 == 0){
            exit(0);
        }
        if (await3 == 0){
            exit(0);
        }
        wait(NULL);
        printf("20 \n");
        if (fork() == 0){ // makes 4 children
            printf("Zwan \n");
            exit(0);
        }
        wait(NULL);
        printf("Zig \n");
    }
}
```

3.2_fork.c

3 Process data communism

Due to personal time management issues, i admit that i copied this exercise from a friend, after we agreed that the solution is likely correct. This was solved in collaboration, i just didn't have the time to reformulate it in my own words.

3.1 Stuffing things in Pipes or cooperative memory use?

Pipes

+	-
- simple synchronisation - easy debugging - easy to implement	- one-to-one communication only - OS driven and therefore not customizable

Shared Memory

+	-
- asynchronous - more than two connections possible - fast	- code can be pretty complex - hard to debug

3.2

Shared Memory:
Hier müsste man ein Segment im Speicher einrichten, welches die Daten aus dem Firefoxfenster nimmt und diese von der Konsole über dieses Segment im Speicher gelesen werden. Dies beinhaltet einen großen Programmieraufwand, da die Kommunikation asynchron abläuft.
Message Passing:
Hier wird eine linked List im Kernel gespeichert, welche die Informationen des Firefox Fensters aufnimmt und dann abspeichert. Diese können von der Konsole ausgelesen werden. Auch diese Kommunikationsform wäre asynchron, aber dafür im System vordefiniert. Außerdem ist diese Kommunikationform Bidirektional, was hier jedoch nicht verwendet wird.
Pipes:
Hier ergeben Pipes am meisten Sinn, da diese synchron arbeiten, keinen Programmieraufwand benötigen, da es in allen Systemen standardisiert ist, und auch unidirektional benötigt werden können. Dies ist praktisch, weil man vom Konsolenfenster keine Daten lesen muss und entsprechend auch keinen Aufwand betreiben muss, diese Datenverbindungen zu timen.

3.3

Ein Thread ist ein Subset eines Prozesses. Dabei kann jeder Prozess einen oder mehrere Threads haben. Dies bezeichnet einfach nur, dass ein Codesegment auf mehreren Threads laufen kann und damit die Execution-Time verringert wird. Da Threads shared Memory besitzen, ist der Austausch zwischen Threads leichter als zwischen isolierten Prozessen.

Erhält ein Prozess mehr Softwarethreads als der Computer Hardwarethreads besitzt, so findet *time slicing* statt. *time slicing* bewirkt, dass jeder Thread einen spezifischen time frame erhält, in welchem er das entsprechende Code-Segment ausführen kann. Dies führt aber zu einem Delay in der Gesamtzeit, weil dann zwischen Threads abgewechselt werden muss, statt sie parallel laufen zu lassen.