TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

Betriebssysteme und Sicherheit, WS 2021/22

3. Aufgabenblatt – Unix III

Bearbeitungszeitraum: 01.11.2021 – 05.11.2021

Ziel dieses Aufgabenblatts ist es die Funktionsweisen von `fork()`, `execve()` und `wait()`, sowie die Beziehungen zwischen den Prozessen nach einer solchen Operation an praktischen Beispielen zu erarbeiten.

Aufgabe 3.1

(a) Erklären Sie, warum nach `fork()` *parent* und *child* zum einen an der gleichen Stelle des Programms fortgesetzt werden, zum anderen dann aber unterschiedliche Wege gehen können.

(b) Welche Ressourcen werden nach `fork()` von *parent* und *child* geteilt und welche werden exklusiv genutzt? Welche Auswirkungen hat das auf den Ablauf von den beiden Prozessen?

(c) Erklären Sie, warum nach `fork()` eine im Hauptspeicher abgelegte globale Variable unterschiedliche Werte in *parent* und *child* haben kann.

Aufgabe 3.2 Entwickeln Sie ein C-Programm, das, ähnlich wie eine *shell*, in einer Endlosschleife auf die Eingabe von Befehlen von einem Nutzer wartet und diese dann als eigenständige Prozesse ausführt. Machen Sie sich dafür nochmals mit der Funktionsweise von `fork()`, `execve()` und `wait()` vertraut. Implementieren Sie das Programm so, dass es nach der Eingabe des Befehls sowohl diesen nochmals in den Nutzer ausgibt als auch die PID des neu gestarteten Prozesses, der den Befehl ausführen soll.

Erarbeiten Sie außerdem den Begriff *Zombie-Prozess* und erläutern Sie, wie er in diesem Zusammenhang entstehen kann.

Aufgabe 3.3 Gegeben sei ein C-Programm, das in seiner `main()` Funktion folgenden Code ausführt:

```
int i, pid;

pid = fork();
if (pid < 0) {
  perror("Error␣during␣fork()");
  exit(1);
}
if (pid) {
  for (i = 0; i < 4; i++)
    puts("parent");
} else {
  for (i = 0; i < 4; i++)
    puts("child");
}

return 0;
```

(a) Kann das gegebene Programmstück zu verschiedenen Ergebnissen führen? Begründen Sie Ihre Antwort und nennen Sie das Ergebnis bzw. einige Ergebnisse. Dabei bewirkt `puts()` die unformatierte zeilenweise Ausgabe der als Parameter angegebenen Zeichenkette.

(b) Wie würde sich die Ausgabe verändern, wenn anstelle auf `stdout` die Zeichenketten in eine *vor dem* `fork()` geöffnete Datei `foo.txt` geschrieben würde?

(c) Wie würde sich die Ausgabe verändern, wenn anstelle auf `stdout` die Zeichenketten in eine *nach dem* `fork()` geöffnete Datei `foo.txt` geschrieben würde?

Klausuraufgaben

Klausuraufgabe I

In einem Unix-ähnlichen System existieren im Verzeichnis `/tmp` die ausführbaren Dateien `Alpha` und `Omega`. Den beiden Programmen liegt folgender C-Code zugrunde:

Alpha

```
1  int main() {
2     int pid;
3     pid = fork();
4
5     if (pid < 0) { printf("F"); }
6     else {
7        if (pid == 0) { exec("/tmp/Omega"); }
8        else          { unlink("/tmp/Omega"); }
9     }
10    printf("A");
11    return 0;
12 }
```

Omega

```
1  int main() {
2     printf("M");
3     return 0;
4  }
```

a) Nennen Sie zwei mögliche Ursachen dafür, dass die Funktion `exec` fehlschlägt.

b) Das Programm `Alpha` wird nun ausgeführt. Der aktuelle Benutzer verfügt dabei über alle erforderlichen Rechte an den beteiligten Dateien und Verzeichnissen. Skizzieren Sie kurz *alle* möglichen Programmabläufe und geben Sie jeweils an, welche Ausgaben dabei auf der Konsole erscheinen.

HINWEIS: Beachten Sie, dass Systemaufrufe fehlschlagen können.

c) Welche Auswirkungen hat es, wenn innerhalb des Quelltextes von `Alpha` in den Zeilen 7 und 8 „Omega“ durch „Alpha“ ersetzt wird?

Klausuraufgabe II

Betrachtet werde das folgende Programm:

```
1  int fd1, fd2, pid1, pid2;
2
3  fd1  = creat("foo.txt");
4  pid1 = fork();
5
6  if (pid1 == 0) {
7     fd2  = creat("bar.txt");
8     write(fd1, "Lorem", 5);
9
10    pid2 = fork();
11    if (pid2 == 0) {
12       write(fd2, "Ipsum", 5);
13       wait();
14    }
15    exit();
16 }
17
18 printf("X");
19 wait();
20 exit();
```

HINWEIS: Die Funktion...

`creat` legt die angegebene Datei an und öffnet sie. Sollte die Datei bereits existieren, wird ihr Inhalt beim Öffnen vollständig gelöscht.

`wait` blockiert den aufrufenden Prozess so lange bis sich ein beliebiger Kindprozess beendet. Existiert kein solcher, so kehrt die Funktion sofort erfolgreich zurück.

`exit` beendet den aufrufenden Prozess sofort.

Ein Nutzer führt das Programm nun aus. Alle auftretenden Funktionsaufrufe sind dabei erfolgreich.

a) Was wurde auf der Konsole ausgegeben unmittelbar nachdem sich der Hauptprozess beendet hat? Skizzieren Sie kurz wie es zu dieser Ausgabe kommt. Sind auch andere Ausgaben möglich? Falls ja, geben Sie die Ursache dafür an und nennen Sie eine zweite mögliche Ausgabe!

b) Welchen Inhalt haben die beiden Dateien `foo.txt` und `bar.txt` unmittelbar nachdem sich der Hauptprozess beendet hat? Gibt es mehrere Möglichkeiten, so geben Sie die Ursache dafür an und nennen Sie einen zweiten möglichen Dateiinhalt!

c) Durch das Entfernen einer einzelnen Zeile des obigen Codes lässt sich erreichen, dass beim Beenden des Hauptprozesses *immer genau drei* `X` ausgegeben wurden. Um welche Zeile (Angabe der Zeilennummer genügt) handelt es sich? Erläutern Sie kurz wieso damit das beschriebene Ergebnis erreicht wird!

d) Welches Ergebnis hätte ein Aufruf von `read(fd1, buf, 10)` – also das Einlesen von bis zu 10 Byte aus `fd1` in den (zuvor angelegten) Puffer `buf` – unmittelbar vor Zeile 20?

Klausuraufgabe III

Beantworten Sie folgende Fragen zu Unix-Prozessen und Shells.

fork.c

```
1  int x = 1;
2  void next() {
3     printf("%d\n", x);
4     x++;
5  }
6  int main() {
7     next();
8     pid_t pid = fork();
9     if (pid > 0) { // Elternprozess
10       wait(NULL);
11       next();
12       next();
13    } else if (pid == 0) { // Kindprozess
14       next();
15    } else { // Fehlerfall
16       next();
17       printf("Fehler\n");
18    }
19    return 0;
20 }
```

a) **Unix-Systemaufrufe:** Welche Ausgabe druckt das obengenannte C-Programm in die Konsole? HINWEIS: Das Einbinden der Header-Dateien sowie ein Teil der Fehlerbehandlung wurden ausgelassen. Gehen Sie von einem fehlerfreien Ablauf aus.

b) **Fehlerbehandlung:** Wir nehmen nun an, dass unmittelbar nach Start des Programms (noch vor dem fork-Aufruf) die maximal erlaubte Prozesszahl des ausführenden Nutzers auf 1 beschränkt wird. Es darf also nur ein einziger Prozess dieses Nutzers gleichzeitig existieren. Geben Sie die Ausgabe an, die in diesem Szenario vom Programm ausgegeben wird.

c) **Unix-Shell:** Geben Sie die Auswirkungen der beiden Befehle „`ls > wc`“ sowie „`ls | wc`“ an und erläutern Sie anhand dessen den Unterschied zwischen den Shell-Operatoren „>“ und „|“.