processTree

Tobias Frank

Jänner 2021

Inhaltsverzeichnis

1 Programmbeschreibung

Das Programm 'processTree' erstellt Prozesse in einer Baumstruktur. Der Benutzer kann angeben wieviele Kinder der Hauptprozess haben soll und wieviele Kinder diese Kindprozesse haben sollen (Blätter des Baumes).
In der Aufgabenstellung stehen zwei Wege wie dieser Baum wieder 'abgebaut' werden soll.
Der erste Weg ist das sich die Prozesse von 'unten' herauf selbst beenden mit dem `quickexit()` Befehl.
Der zweite Weg ist die Beendigung mit einem Signal, dass vom Hauptprozess (Wurzel des Baumes) ausgeht. Der schickt ein Signal zum Beenden an seine Kinder. Die Kinder erhalten das Signal und leiten es an ihre Kinder, also die Blätter, weiter. Wenn das Signal dort angekommen ist, beenden sich die Kinder.

Die Aufgabenstellung wurde meinerseits so erweitert, dass der Benutzer zusätzlich angeben kann welche 'Zerstörungsmethode' gewählt werden soll und, dass beim Beenden per Signal die Kindprozesse auch beendet werden, nachdem sie alle jeweils zugehörige Blätter beendet haben.
Zusätzlich habe ich ergänzt, dass es einen 'safe - mode' gibt. In diesem befindet man sich standardmäßig. Hier ist es nur maximal möglich 5 Kinder mit jeweils 5 Blättern zu erstellen. Sollte es Benutzer geben die ihren Rechner und das Programm auf ihre Grenzen testen wollen kann beim Start des Programmes ein Parameter mitgegeben werden (siehe Kap. 2 Bedienung).

1.1 Prozesse in C++

Ein Prozess wird in C++ mittels

```
fork()
```

dupliziert. Dies geschieht inklusive Puffer. Der Kindprozess ist ab dem Zeitpunkt des forkens ein Duplikat des Vaterprozesses.
Dieses Duplikat bekommt dann als Rückgabewert von `fork()` den Wert 0.

Damit kein Zombieprozess entstehen kann muss der Elternprozess `waitpid()`aufrufen.

Ein Codeaussschnitt zum Verdeutlichen wie in C++ Prozesse erstellt werden.

```
int main() {
    pid_t pid;
    pid = fork();
    if (pid == -1) {  //Fehler bei der fork() - Funktion
        cerr << "forking failed: " << errno<< endl;
        exit(EXIT_FAILURE);
    }
    if (pid == 0) {   //hier befindet man sich im Kindprozess
```

```
        ...

    } else {          //hier befindet man sich im Vaterprozess
        ...
        int status;
        waitpid(pid, &status, 0);
        exit(EXIT_SUCCESS);
    }
}
```

Quelle: http://openbook.rheinwerk-verlag.de/linux_unix_programmierung/Kap07-007.htm, 10.1.2021

1.2 Signale in C++

Signale sind ein einfacher Weg um zwischen Prozessen kommunizieren zu können.

1.2.1 Mit Signalen umgehen

Um mit Signalen umgehehen zu können muss die Funktion `signal_handler()` implementiert werden.
Diese bekommt als Parameter die Nummer des Signals. In der Funktion kann entschieden werden wie mit diesem Signal umgegangen wird.

```
void signalHandler(int sinal) {
   cout <<  signal <<  endl;
   ...
}
```

Quelle: https://www.tutorialspoint.com/cplusplus/cpp_signal_handling.htm, 10.1.2021

1.3 Programmablauf

Der Programmablauf startet mit der Übergabe von 2 Werten durch den Benutzer. 'm' als Anzahl der Kinder und 'n' als Anzahl der Blätter. Somit startet beispielsweise ein Aufruf mit

`./processTree 3 2`

3 Kinder, die dann jeweils 2 Blätter starten.

Zusätzlich kann er mittels '–k' angeben, dass der Baum mittels ´Signalen abgebaut werden soll.
Angeommen er gibt diese Option an kommt man in die Funktion

```
void fork_process(int n, int m, int depth) {
  int i = 0;
  if (depth == 0) {
    while (i < n) {
      auto pid{fork()};
      if (pid == -1) {
        cerr << "forking failed : " << errno << endl;
        exit(EXIT_FAILURE);
      }
      if (pid == 0) {
        cnt1 += i;
        fmt::print(fg(fmt::color::red), "{0} \n", cnt1);
        print_process();
        fork_process(n, m, depth + 1); //Funktionaufruf -> Rekursion!
        sleep(50);
      }
      else {
        children.push_back(pid);
      }
      sleep(1);
      i++;
    }
    sleep(2); //Hauptprozess nachdem der Baum fertig aufgebaut ist
    fmt::print(fg(fmt::color::gold), "Finished creating tree. \n \n \n");
    logger->info("created tree");
    int status;
    for (size_t k = 0; k < children.size(); k++) {
      fmt::print(fg(fmt::color::light_blue), "Sending Signal to: {0}", children[k]);
      kill(children[k], SIGTERM); //signal_handler() wird aufgerufen.
      waitpid(children[k], &status, 0);
    }
  }
  else if (depth == 1) {  //Funktionsaufruf mit depth = 1! -> Blätter werden erstellt
    while (i < m) {
      auto pid{fork()};
      if (pid == -1) {
        cerr << "forking failed : " << errno << endl;
        exit(EXIT_FAILURE);
      }
      if (pid == 0) {
        cnt2 += i;
        fmt::print(fg(fmt::color::red), "{0}.{1} \n", cnt1, cnt2);
        print_process();
        sleep(50);
      }
```

```
      else {
        children.push_back(pid);
      }
      i++;
    }
    sleep(1);
  }
  else
  {
    return;
  }
}
```

Wie man erkennen kann, verläuft der Ablauf des Prozesse Erstellens mittels einer Kombination einer while-Schleife und Rekursion. Der Parameter `depth` gibt die Tiefe an auf der man sich in Programm gerade befindet. Der erste Durchlauf startet mit `depth`= 0. Jetzt werden in einer while - Schleife die Kinder erstellt. In diesen Kindprozessen wird die Funktion nochmals aufgerufen, diesmal jedoch mit `depth` = 1. Hier werden die Blätter der Kinder in einer Schleife erstellt. Dann beendet sich dieser Funktionsaufruf und das nächste Kind wird erstellt. Dort wird dann wieder ein Funktionsaufruf gemacht für die Blätter. Dieser Vorgang wird solange wiederholt bis alle gewünschten Blätter und Kinder erstellt sind. Dann wird 2 Sekunden gewartet. Darauf wird an jedes Kind ein Signal gesendet. Somit wird der `signal_handler` aufgerufen. Dieser geht die Kinder der Kinder durch und sendet an jedes einzelne ein Signal zum Beenden.

```
void signalHandler(int signal) {
 if(signal == 15){
  if (children.size() > 0){
    for(size_t c = 0; c < children.size(); c++){
      fmt::print(fg(fmt::color::green), "Process {1} here, got Signal,
      delegating it to leaf {0} \n \n", children[c], getpid());
      kill(children[c], SIGKILL); //sendet Signal an Blatt zum beenden
      fmt::print(fg(fmt::color::red), "{} killed because of
      signal \n \n", children[c]);
    }
  }
  fmt::print(fg(fmt::color::gold), "Finished killing leaves per signal. \n");
  fmt::print(fg(fmt::color::red), "-> Now killing child {} \n \n \n", getpid());
  quick_exit(EXIT_SUCCESS);
  } else {
    cout << "Got bad signal!" << endl;
    return;
  }
}
```

Nachdem alle Blätter beendet sind, beendet sich das Kind auch.

1.4 Verwendete Bibliotheken

- `CLI11` für Kommandozeilenargumente
- `fmt` für Formatierung der Ausgaben im Terminal
- `spdlog` für Logging

2 Bedienung

Das Programm wird aufgerufen mittels:

```
./processTree m n (--killsignal od. --k) (--log od. --l)
```

Hier bei steht m für die Anzahl der Kinder und n für die Anzahl der Blätter.

Mit `--k` oder `--killsignal` kann angegeben werden, dass die Prozesse per Signal beendet werden sollen.

Mit `--l` oder `--logs` kann angegeben werden, ob Logs in eine Datei geführt werden sollen oder ob keine Logs geführt werden sollen.

Mit `--e` oder `--experimental` kann angegeben werden, ob man den safe - mode verlassen möchte. Dann können die Parameter m und n größer als 5 sein.