Übungsblatt 2 zur Vorlesung Parallele Programmierung

Abgabe: 02.11.2024, 23:59

Prof. Dr. Michael Kuhn (michael.kuhn@ovgu.de)
Michael Blesel (michael.blesel@ovgu.de)
Parallel Computing and I/O • Institut für Intelligente Kooperierende Systeme
Fakultät für Informatik • Otto-von-Guericke-Universität Magdeburg
https://parcio.ovgu.de

Um später effizient programmieren zu können, wollen wir uns ein wenig mit der Fehlersuche beschäftigen. Hierzu schauen wir uns den GNU Debugger GDB und den Speicherprüfer memcheck aus dem Valgrind-Projekt an. Diese können sowohl für serielle als auch für parallele Programme genutzt werden. In C ist die Speicherallokation sehr fehlerträchtig, Valgrind ist hierbei sehr hilfreich, um typische Fehler aufzuspüren.

1. Debugging einer einfachen Anwendung (150 Punkte)

Im Verzeichnis simple ist ein primitives Programm enthalten, welches mit make kompiliert werden kann. Dieses Programm dient dazu, dass Sie sich ein wenig mit GDB und Valgrind beschäftigen. Es enthält lediglich vier Funktionen, welche jeweils einen Zeiger auf eine Zahl oder ein Array mit einer Zahl enthalten und gibt diese dann in der main-Funktion aus. Leider enthält dieses Programm diverse Fehler.

- Führen Sie folgende kleinere Tests durch, um GDB kennen zu lernen. Dokumentieren Sie die genutzten Eingabebefehle und die Ausgabe in einer Textdatei. Ein kurzes Tutorial für GDB finden Sie beispielsweise unter https://www.cs.cmu.edu/~gilpin/tutorial/. Die wichtigsten Befehle sind außerdem in https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf zusammengefasst.
  - Starten Sie das Programm. Dafür übergeben Sie am besten direkt den Programmnamen an GDB (gdb ./simple).
  - Platzieren Sie einen Breakpoint auf der Funktion mistake1, starten Sie das Programm, geben Sie den Wert von buf und buf[2] aus. Gehen Sie zur nächsten Zeile und geben Sie beide Werte wieder aus. Von welchem Typ ist buf?
  - Platzieren Sie einen Breakpoint in der Funktion mistake2, setzen Sie den Programmlauf fort, welchen Typ hat buf?
  - Setzen Sie den Programmlauf fort, welcher Text wird ausgegeben? Lassen Sie sich den Code um diese Stelle herum ausgeben. Welche Frames sind auf dem Stack? Wechseln Sie zu Frame 1. Geben Sie den Inhalt von p aus.
  - Rufen Sie innerhalb von GDB die Funktion mistake4 auf (schauen Sie nach, wie man in GDB Funktionen direkt aufrufen kann).
- Modifizieren Sie das Programm zunächst so, dass es nicht mehr abstürzt. Versuchen Sie die Modifikationen möglichst gering zu halten. Verwenden Sie zunächst GDB, um die Fehlerstellen aufzuspüren. Die Ausgabe soll wie folgt aussehen:

```
1: 1
2: 2
3: 3
4: 4
```

- Nun läuft das Programm, leider enthält es jedoch noch weitere Speicherfehler, die je nach Umgebung (mehr oder weniger zufällig) auftreten können. Modifizieren Sie das Programm unter Zuhilfenahme von Valgrinds `memcheck` so, dass jede Methode Speicher korrekt reserviert und dass am Ende der Speicher korrekt freigegeben wird. Rufen Sie dafür das Programm mit `valgrind ./simple` auf.

  Hinweis: Den Speicher einfach mit `static` zu allokieren ist *nicht* erlaubt. Verzichten Sie darüber hinaus auf globale Arrays und Variablen. Eine kurze Anleitung für Valgrind finden Sie beispielsweise unter `https://www.valgrind.org/docs/manual/quick-start.html`.

Dokumentieren Sie die Fehler, die zu den Abstürzen und Speicherfehlern führen. Notieren Sie hierfür für jeden vorhandenen Fehler die Code-Zeilen, welche fehlerhaft sind und den genauen Grund der Ursache (z. B. Speicher mehrfach freigegeben).

2. Debugging einer komplexen Anwendung (120 Punkte)

Im Verzeichnis `pde` finden Sie ein numerisches Programm zum Lösen von Differentialgleichungen. Grundsätzlich ist das Programm vom Ablauf korrekt, jedoch haben sich durch Unachtsamkeit einige Flüchtigkeitsfehler u. a. bei der Speichernutzung eingeschlichen. Um das Programm zu korrigieren, müssen Sie nicht genau verstehen was berechnet wird, u. U. sollten Sie jedoch mit einem Debugger die Aufrufe ein wenig verfolgen. Korrigieren Sie alle Fehler im Programm. Modifizieren Sie hierbei den Code so wenig wie nötig.

Das Programm sollte dabei wie folgt aufgerufen werden: `./partdiff 1 1 100 2 5`

3. Batch Scheduling (45 Punkte)

Beantworten Sie die folgenden Fragen und notieren sie gegebenenfalls die Befehlsausgaben:

1. Was bedeuten die Begriffe Job Scheduler bzw. Batch Scheduling im Bezug auf einen Cluster?
2. Welche Aufgaben hat ein Job Scheduler?
3. Welcher Job Scheduler wird auf dem Cluster verwendet?
4. Machen Sie sich mit der Manpage von `sbatch` vertraut. Beschreiben Sie die Funktionsweise des Kommandos.
5. Wie lassen sich die aktuellen Jobs und deren Status anzeigen?
6. Gibt es eine Möglichkeit, einen bereits abgeschickten Job zu löschen (bevor oder während er läuft)? Wenn ja, wie?

7. Welche unterschiedlichen Partitionen sind auf dem Cluster eingerichtet? Wie kann die zu benutzende Partition geändert werden?
8. Finden Sie heraus, wie Sie einen einzelnen Knoten allokieren können. Allokieren Sie einen konkreten Knoten (z. B. `ant13`), bauen Sie eine Verbindung zu diesem Knoten auf und führen sie `hostname` darauf aus.
9. Wie hoch ist das Zeitlimit auf dem Cluster, bis alle Knoten wieder freigegeben werden?

Abgabe

Als Abgabe werten wir den letzten Commit vor der Abgabefrist in Ihrem Git-Repository. Im Hauptverzeichnis des Repositories wird ein Verzeichnis `PP-2024-Uebung-02-Materials` mit folgendem Inhalt erwartet:

1. Eine Datei `gruppe.md` mit den Gruppenmitgliedern (eines je Zeile) im folgenden Format:

   ```
   Erika Musterfrau <erika.musterfrau@example.com>

   Max Mustermann <max.mustermann@example.com>
   ```

2. Eine Textdatei mit den Ein-/Ausgaben von GDB mit dem Namen `simple-gdb.md`, eine Textdatei namens `simple-error.md` mit Fehlerbeschreibungen (Ursache, Code-Zeilen) und der überarbeitete Code im Verzeichnis `simple` (Aufgabe 1)
3. Eine Textdatei `pde-error.md` mit Fehlerbeschreibungen (Ursachen, Code-Zeilen) und der überarbeitete Code im Verzeichnis `pde` (Aufgabe 2)
4. Eine Textdatei `antworten.md` mit Ihren Antworten (Aufgabe 3)