Aufgabe 1: Einführung in Git und C

Bevor Sie die folgenden Aufgaben bearbeiten, schauen Sie sich bitte die Folien der großen Übung an und beschäftigen Sie sich mit der Programmiersprache C, dem Kompiliervorgang sowie dem Schreiben von Make-Skripten. Das Ziel der folgenden Aufgaben ist es, den grundlegenden Umgang mit der Softwareentwicklung in C und den gängigsten Werkzeugen zu üben.

1.1 Git (5 Punkte)

Die Übungsaufgaben der Vorlesung Betriebssysteme werden über das verteilte Versionsverwaltungssystem *Git* organisiert, das Sie bereits aus der Tafelübung kennen. In der ersten Aufgabe sollen Sie sich mit dem Kommandozeilentool `git` und der Anwendung *GitLab* vertraut machen. GitLab ist eine quelloffene Webanwendung, welche den Umgang mit Git-Repositories vereinfacht und zusätzliche Features wie Nutzermanagement, Bug-/Issue-Tracking und Continuous Integration (CI) bietet. Das IBR betreibt eine eigene GitLab Instanz unter `https://gitlab.ibr.cs.tu-bs.de/`, welche im Rahmen dieser Übung genutzt wird. Führen Sie die Befehle in der angegebenen Reihenfolge aus. Weitere Informationen zu den einzelnen Kommandos erhalten Sie mit `man git-[command]` oder `git help [command]`.

1.1.1 Git Konfiguration

Im Folgenden sollen Sie Ihre Git Umgebung grundlegend konfigurieren. Öffnen Sie dazu ein Terminal (bspw. *Gnome Terminal*) und führen die angegebenen Befehle aus. Die Konfigurationsparameter werden in der Datei `~/.gitconfig` gespeichert.

- Hinterlegen Sie *Ihren* Namen mittels
  `git config --global user.name "Max Mustermann"`
- Hinterlegen Sie *Ihre* TU E-Mail-Adresse mittels
  `git config --global user.email m.mustermann@tu-bs.de`
- Konfigurieren Sie ein standardmäßiges *rebasing* bei `git pull` (vgl. Tafelübung):
  `git config --global pull.rebase true`
- Setzen Sie `meld` als Mergetool für Git:
  `git config --global merge.tool meld`
- Weisen Sie Git an, lokale Branches standardmäßig auf entfernte Branches gleichen Namens zu pushen:
  `git config --global push.default current`
- Wiederholen Sie diese Schritte für jedes Teammitglied mit der entsprechenden y-Nummer. *Wichtig:* Loggen Sie sich dazu auch neu am Rechner ein, um zu verhindern, dass die Konfigurationsdateien überschrieben werden!

1.1.2 GitLab Zugang einrichten

Der komfortabelste Weg, um mit Git Repositories zu interagieren, ist das SSH Protokoll. Zur Authentifizierung muss dazu *einmalig* ein öffentlicher SSH Schlüssel für jeden Nutzer im GitLab hinterlegt werden. Folgen Sie dazu den unten angegeben Schritten.

- Loggen Sie sich im Raum G40 im Informatikzentrum an einem der Rechner mit Ihrem y-Account ein.
- *Führen Sie diesen Schritt nur durch, wenn Sie noch kein SSH Schlüsselpaar erzeugt haben!* Öffnen Sie ein Terminal und geben Sie `ssh-keygen` ein und drücken Sie drei mal Enter. Die Passphrase *können* Sie leer lassen.
- Geben Sie `cat .ssh/id_rsa.pub` ein, um den öffentlichen Teil ihres neu generierten Schlüsselpaares anzuzeigen. Kopieren Sie die komplette Ausgabe von `ssh-rsa` bis `@izXX` in die Zwischenablage.
- Gehen Sie mit einem Browser auf `https://gitlab.ibr.cs.tu-bs.de/`und loggen sich über den Reiter „LDAP“ mit Ihrer y-Nummer ein. Klicken Sie oben rechts auf das Benutzermenü → Settings → SSH-Keys und fügen Sie den neu generierten Schlüssel hinzu.
- Wechseln Sie zurück ins Terminal und geben Sie `ssh git@gitlab.ibr.cs.tu-bs.de` ein. Bestätigen Sie die eventuelle Nachfrage `Are you sure you want to continue connecting (yes/no)?` mit der Eingabe von `yes`. Wenn Sie mit der Ausgabe `Welcome to GitLab, ...` gefolgt von einem `Connection to ... closed.`

begrüßt werden, haben Sie den Zugang erfolgreich eingerichtet. Falls nicht, suchen Sie nach möglichen Fehlern in den vorigen Schritten. Nutzen Sie dazu auch die offizielle Dokumentation von GitLab zum Thema SSH Keys[1].

- Wiederholen Sie diese Schritte für jedes Teammitglied mit der entsprechenden y-Nummer. *Wichtig:* Loggen Sie sich dazu auch neu am Rechner ein, um zu verhindern, dass die gerade generierten SSH-Schlüssel überschrieben werden.

1.1.3 Klonen des Git-Repositories

In GitLab können mehrere Git-Repositories zu Gruppen („groups“) zusammengefasst werden, um Repositories mit ähnlichen Inhalten zu organisieren. Die Gruppe ws1819-bs-studs beinhaltet alle studentischen Repositories zur Vorlesung.

- Finden Sie im GitLab die Gruppe „ws1819-bs-studs“.
- Finden Sie innerhalb der Gruppe „ws1819-bs-studs“ das Ihrem Team zugeordnete Repository und kopieren Sie dessen URL der Form `git@gitlab.ibr.cs.tu-bs.de:ws1819-bs-studs/[name].git`. *Wichtig:* Nutzen Sie immer die SSH URL, nie die HTTPS URL! Den Namen Ihres Repositories entnehmen sie der E-Mail, die Sie bekommen haben. Sollten Sie keine E-Mail bekommen haben wenden Sie sich bitte an einen Übungsbetreuer.
- Klonen Sie das Repository mit `git clone [URL]` und wechseln Sie in das neu angelegte Verzeichnis `[name]` mit `cd [name]`.

1.2 Umgang mit dem Git-Repository

Git wird üblicherweise mit dem zugehörigen Kommandozeilenprogramm `git` bedient. `git` erwartet dabei als erstes Argument ein Kommando, wie beispielsweise `clone` aus der vorherigen Aufgabe. Eine Übersicht aller verfügbarer Kommandos finden Sie in der *Manpage* von `git`, die Sie mit dem Befehl `man git` anzeigen lassen können. Ausführliche Beschreibungen für jedes einzelne Kommando erhalten Sie mit `man git-[command]` oder `git help [command]`, also beispielsweise `man git-clone` bzw. `git help clone`.

- Verschaffen Sie sich einen Überblick über die von `git` unterstützten Kommandos auf der Manpage von `git`. Wichtig dabei sind vor allem die Kommandos auf hoher Ebene ("porcelain commands").
- Legen Sie eine neue Datei namens `file1` und schreiben Sie beliebigen Inhalt in *die ersten drei Zeilen* der Datei. Geben Sie `git status` ein um den aktuellen Stand des Git Repositories anzuzeigen. Die Datei `file1` wird unter `Untracked files` aufgelistet, fügen Sie deshalb die Datei mit `git add file1` zum Git Index hinzu. Vergewissern Sie sich mit einem erneuten Aufruf von `git status` davon, dass die Datei erfolgreich hinzugefügt wurde.
- Führen Sie Ihren ersten Commit durch: Geben Sie `git commit` ein, um Änderungen an Dateien im Index der *Git History* hinzuzufügen. Standardmäßig öffnet sich der Editor `nano` und verlangt die Eingabe einer *Commit Message*, die Ihre Änderungen beschreibt. Geben Sie eine entsprechende Nachricht ein und speichern Sie die Datei. Alternativ können Sie `git commit -m` verwenden. Vergewissern Sie sich mittels `git status` davon, dass Ihr Repository keine weiteren Änderungen enthält (Ausgabe: `nothing to commit, working tree clean`). Laden Sie die Änderungen mit `git push` auf den GitLab Server hoch.
- Loggen Sie sich aus und wechseln Sie den Benutzer am Rechner im G40: Lassen Sie ein *anderes* Teammitglied ein `git clone` Ihres Repositories ausführen. Editieren Sie die Datei `file1` und verändern Sie die *zweite Zeile* der Datei. Führen Sie analog zum vorherigen Schritt `git status`, `git add`, `git commit` und `git push` aus. Ändern Sie nun die *dritte Zeile* der Datei `file1`, comitten Sie Ihre Änderung, aber führen Sie *kein* `git push` aus, um parallele Änderungen simulieren.
- Loggen Sie sich wieder aus und wechseln Sie den Nutzer erneut. Führen Sie `git pull` im Git Repository aus, um die gemachten Änderungen vom GitLab Server herunterzuladen. Ändern Sie in `file1` die *erste Zeile* und committen und pushen Sie Ihre Änderungen.
- Loggen Sie sich aus und wechseln Sie zum vorherigen Nutzer, führen Sie im Repository zunächst ein `git pull` aus. Git lädt die neuen Änderungen herunter und wendet die nicht gepushten Änderungen darauf an. Da immer verschiedene Zeilen der Datei geändert wurden, kann Git die beiden Versionen von `file1` konfliktfrei automatisch zusammenführen (Ausgabe: `Auto-merging file1`). So wird der ältere Commit auf den jüngeren angewendet. Vergewissern Sie sich davon durch die Eingabe von `tig`, welche eine Übersicht aller Commits anzeigt.

[1] `https://docs.gitlab.com/ce/ssh/`

- Provozieren Sie einen *Merge Konflikt*, indem Sie mit mehreren Nutzern parallel die gleiche Zeile von `file1` bearbeiten und ihre Änderungen pushen. Führen Sie erneut `git pull` aus. Einen Merge Konflikt erkennen Sie an der Ausgabe `CONFLICT (content): Merge conflict in file1`. Dieser Konflikt kann von Git nicht automatisch gelöst werden und erfordert Ihr Eingreifen. Starten Sie `meld` mittels `git mergetool` und beheben Sie den Konflikt, indem Sie die in der Mitte angezeigte Version von `file1` entweder auf die lokale Version (links) oder entfernte Version (rechts) bringen. Nutzen Sie dazu die Buttons mit den Pfeilen. Speichern Sie Ihre Änderung und verlassen Sie `meld`. Bestätigen Sie Ihre Änderung mit `git add file1` und commiten und pushen Sie Ihre Konfliktlösung.
- Löschen Sie die Datei `file1` mit `git rm file1` aus dem Git Index und committen und pushen Sie diese Änderung.

1.3 Einfache Makefile (3 Punkte)

Öffnen Sie die Datei `hello.c` und schreiben Sie darin ein einfaches C-Programm, dass „Hello World“ ausgibt. Schreiben Sie außerdem passend dazu eine Makefile, wie es in der großen Übung beschrieben wird, welche die Targets `all`, `clean` und `helloworld` bereitstellt. Vergessen Sie nicht, beide neuen Dateien mit `git add` der Versionierung hinzuzufügen.

- `helloworld` soll den Quellcode in `hello.c` zum Programm `helloworld` kompilieren. Verwenden Sie dafür den Compiler **clang(1)** mit den Parametern: **-std=c11 -pedantic -Wall -Wextra**
- `all` soll das Target `helloworld` erzeugen.
- `clean` soll mit Hilfe von **rm(1)** alle .o-Dateien sowie die `helloworld`-Datei aus dem Arbeitsverzeichnis entfernen. Dieses Target sollte immer fehlerfrei funktionieren, auch wenn z.B. die Datei `helloworld` nicht existiert.

Beachten Sie, dass für einige dieser Targets ein Eintrag im .PHONY Target nötig ist (s. große Übung). *Hinweis:* Fügen Sie auf keinen Fall generierte Objekte wie die .o-Dateien oder das fertig kompilierte Programm zur Versionierung hinzu! Diese Dateien sollten nicht im Git Repository sein.

1.4 Generische Makefile Targets (2 Punkte)

Erweitern Sie Ihre Makefile um ein generisches Target, das eine beliebige .c-Datei in eine gleichnamige .o-Datei kompiliert. Compiler und Flags sollen identisch zum vorherigen Aufgabenteil sein. Ein Verlinken der Dateien soll dabei nicht erfolgen. Dieses erreichen Sie mit dem Compilerparameter `-c`. Folgende Liste an Wildcard-Patterns kann Ihnen dabei helfen:

- Wildcards für Targets und Abhängigkeiten
  - `%`: Name des an `make` übergebenen Targets
- Wildcards für Erzeugungsregeln
  - `$@`: Name des Targets
  - `$<`: die erste Abhängigkeit
  - `$+`: eine Liste aller Abhängigkeiten
  - `$^`: eine Liste aller Abhängigkeiten, doppelte Abhängigkeiten werden eliminiert

Testen Sie das generische Target, indem Sie die Dateien `hello.o`, `shellutils.o` (die eine Sammlung von Hilfsfunktionen enthält) und `test.o` (ein kleines shellutils Testprogramm) damit erzeugen. Passen Sie das `helloworld` Target an, sodass hier nur noch der Linker aufgerufen wird. Die eigentliche Kompilierung des Codes soll über das generische Target erfolgen. Passen Sie außerdem den Compiler- bzw. Linker-Aufruf so an, dass die Eingabedateien hier aus den Target-Abhängigkeiten automatisch erzeugt werden. Die Aufgabe ist erfolgreich gelöst, wenn alle drei .o-Dateien erstellt werden können und `helloworld` unter Verwendung des generischen Targets erzeugt wird. **Für die Lösung dieser Aufgabe dürfen keine impliziten Make-Targets verwendet werden.**

1.5 Debugging (5 Punkte)

- Erweitern Sie die Makefile um ein Target namens `shellutilstest`, das die Dateien `shellutils.o` und `test.o` zu einer ausführbaren Datei namens `shellutilstest` linkt. Erweitern Sie außerdem das `all`-Target um das neue Target und passen Sie auch das `clean`-Target so an, dass beim Aufruf auch die `shellutilstest`-Datei entfernt wird.
- Verlinken Sie `shellutils.o` und `test.o` zu einer ausführbaren Datei und führen Sie das Programm aus.
- Unerfreulicherweise stürzt das Programm mit einer wenig hilfreichen Fehlermeldung ab. Ohne sinnvolle Hinweise, wo der Fehler liegen könnte, wird es sehr schwierig, den Fehler zu finden. Eine Möglichkeit, den Fehler zu lokalisieren, wäre es, Debug-Ausgaben in den Quelltext einzubauen, aber auch damit benötigt man meistens viel Zeit, bis der Fehler gefunden werden kann. Schneller geht es oft, wenn man einen Debugger zu Hilfe nehmen kann. Bisher enthält unsere ausführbare Datei jedoch keinerlei Informationen über den Quellcode für den Debugger. Folglich kann der Debugger aus der Binärdatei keinen Bezug zu unserem Quellcode herstellen. Um das zu ändern, müssen wir beim Kompilieren Hinweise für den Debugger einbauen. Dies geschieht, indem man dem Compiler den Parameter `-g` übergibt. Zusätzlich sollte man dem Compiler für das Debuggen den Parameter `-O0` übergeben, um jegliche Optimierungen abzuschalten, weil diese die Debugger-Ausgaben verfälschen können.
- Räumen Sie mit Hilfe von `make clean` Ihr Arbeitsverzeichnis auf, passen Sie die Compilerparameter an und kompilieren Sie den `shellutilstest` erneut.
- Verwenden Sie zum Debuggen das Programm **cgdb(1)**. Dies ist eine grafisch aufbereitete Version des GNU Debuggers (**gdb(1)**). Starten Sie Ihr Programm mit Hilfe des Debuggers über `cgdb shellutilstest`. Lassen Sie das Programm laufen, bis der sogenannte *Segmentation Fault* erneut eintritt.
- Jetzt bekommen Sie eine ziemlich genaue Information, wo das Programm abgestürzt ist. *Markieren Sie diese Zeile im Code mit einem Kommentar.* Sollte dies einmal nicht der Fall sein, hilft Ihnen der Befehl `backtrace`, um herauszufinden, welche Funktionsaufrufe zum aktuellen Zeitpunkt aufgerufen, aber noch nicht beendet wurden.
- Sollten Sie die Ursache des Absturzes sehen, können Sie jetzt den Fehler beheben. Wenn Sie den Fehler noch nicht sehen oder noch etwas mit **cgdb(1)** üben wollen, sollten Sie einen Breakpoint auf die Funktion setzen, in der es zum Absturz gekommen ist. Starten Sie dann Ihr Programm mit `run` im Debugger neu. Das Programm läuft jetzt, bis es zum ersten Mal den Breakpoint passiert, und hält dort an. Arbeiten Sie sich von diesem Punkt aus mit den Befehlen `step`, `next` und `finish` durch den Code bis kurz vor die Stelle, an der es zum Absturz kommt. Versuchen Sie dabei, ein wenig nachzuvollziehen, was der Programmcode tut. Schauen Sie sich dann mit dem Befehl `print` verdächtige Variablen an. Sobald Sie die Ursache des Fehlers gefunden haben, beheben Sie diesen und committen Sie Ihre Codeänderungen.
- Die Aufgabe ist erfolgreich abgeschlossen, wenn die fehlerhafte Zeile markiert wurde und das Programm `shellutilstest` fehlerfrei ausgeführt werden kann.

1.6 Abgabe

Committen und pushen Sie alle modifizierten Dateien in Ihr Git Repository. Anschließend können Sie sich wie auf dem Blatt mit weiteren Hinweisen zur Rechnerübung beschrieben Feedback über automatisierte Tests geben lassen.

Abgabe bis 26. Oktober, 23:59 Uhr.

Ihre Lösungen sollen bis zur Deadline im GitLab des IBR hochgeladen sein.
Einhaltung der Coding Guidelines: 0 Punkte
Insgesamt zu erzielende Punktzahl: 15 Punkte
Zu erzielende Minimalpunktzahl: 8 Punkte