182.709 Betriebssysteme UE
Demo-Test

20. November 2020

Matrikelnummer

Zuname, Vorname

Punkte (30)

Testumgebung

Im Verzeichnis `~/exam` finden Sie die C-Quelldateien, in welchen Sie Ihre Lösung implementieren sollen. Sie können jederzeit die Original-Dateien mit dem Shellkommando

```
$ fetch
```

wiederherstellen. Änderungen, die Sie an den Dateien vorgenommen haben, werden gesichert (`<Datei>`→`<Datei>.~1~`).

In dem Verzeichnis befindet sich auch ein Makefile. Um Ihre Lösung zu kompilieren, verwenden Sie den Befehl `make`. Testen und debuggen Sie Ihr Programm wie gewohnt. Auch `gdb` steht Ihnen zur Verfügung. Übrig gebliebene Ressourcen (wie Semaphoren oder Shared Memory) können Sie mit dem Kommando `cleanup` entfernen.

Um Ihre Lösung zu prüfen, führen Sie

```
$ deliver
```

aus. Sie können `deliver` beliebig oft ausführen. Dabei wird Ihnen auch angezeigt, wie viele Punkte Sie derzeit auf Ihre Lösung erhalten würden.

Wenn Sie mit der Bewertung zufrieden sind, dann melden Sie sich bei der Aufsicht, die das Ergebnis entgegennimmt und Sie ausloggen wird.

Der Test besteht aus mehreren Aufgaben, die Sie **unabhängig voneinander** und in **beliebiger Reihenfolge** implementieren können. Für jede Aufgabe steht Ihnen eine DEMO-Funktion zur Verfügung, die Sie in der `main()`-Funktion einkommentieren können, wenn Sie eine Aufgabe überspringen möchten.

Beachten Sie folgende Bewertungskriterien:

- Ihr Programm muss ohne Fehler kompilieren, sonst erhalten Sie keine Punkte.
- Compiler Warnings (das Programm wird mit `-Wall` übersetzt) führen zu Punkteabzügen.
- Bei Segmentation Faults (Speicherzugriffsverletzung) erhalten Sie für die entsprechende Aufgabe keine Punkte.

Sie können den Prüfungsbogen für Notizen verwenden. Diese Eintragungen werden nicht bewertet.

Einleitung

Schreiben Sie ein Programm, welches ein Passwort übergeben bekommt, dieses in einen geteilten Speicher schreibt und anschließend ein Server-Programm anweist, für dieses Passwort einen Hash zu berechnen. Sobald der Server damit fertig ist, soll Ihr Programm den generierten Hash wieder aus dem gemeinsamen Speicher lesen. **Das Programm für den Server ist bereits vorgegeben. Ihre Aufgabe ist es, den dazugehörigen Client zu implementieren.**

Die Kommunikation zwischen Ihrem Client und dem Server erfolgt ausschließlich über einen gemeinsamen Speicher (en.: shared memory) und wird mittels benannter Semaphore (en.: named semaphores) synchronisiert. Für die Synchronisation werden folgende 3 Semaphore verwendet:

- `sem_request`: Diese Semaphore verwendet ein Client, um dem Server mitzuteilen, dass eine Anfrage (en.: request) in den gemeinsamen Speicher geschrieben wurde.
- `sem_response`: Diese Semaphore verwendet der Server, um einem Client mitzuteilen, dass die Antwort (en.: response) auf seine Anfrage in den gemeinsamen Speicher geschrieben wurde.
- `sem_client`: Diese Semaphore dient mehreren Clients als wechselseitiger Ausschluss (en.: mutual exclusion), um immer nur einem Client zugleich die Kommunikation mit dem Server zu erlauben.

Untenstehender Pseudocode zeigt, wie ein Kommunikations-Zyklus zwischen Ihrem Client und dem Server abgewickelt wird. Der Server versucht zunächst, die Semaphore `sem_request` zu dekrementieren und blockiert solange das nicht möglich ist. Sobald `sem_request` vom Server erfolgreich dekrementiert wurde, wir das als erfolgte Anfrage eines Client gewertet. Der Server liest daraufhin den Inhalt des gemeinsamen Speichers und berechnet dafür einen Hash. Dieser generierte Hash wird dann zurück in den gemeinsamen Speicher geschrieben. Der Server zeigt an, dass die Bearbeitung des Passworts und Generierung des Hash abgeschlossen ist, indem die Semaphore `sem_response` inkrementiert wird. Anschließend wartet der Server auf weitere Anfragen.

Client	**Server**
... client makes a request; ...	

	`wait(sem_request);`
	process request;
	write response to shared memory;
	`post(sem_response);`

... client reads the response; ...	

Pseudocode eines Kommunikations-Zyklus zwischen einem Client und dem Server; die nötige Synchronisation auf Seite des Servers fehlt hier noch

1 Argumentebehandlung (10)

Vervollständigen Sie die Funktion `void parse_arguments(int argc, char *argv[], args_t *args)` in `client.c` mit der Argumentebehandlung.

Synopsis

```
./client -p PASSWORD
```

Der Client akzeptiert `-p` als einzige Option, mit der ein Passwort übergeben wird. Diese Option muss genau einmal vorkommen.

Positionelle Argument sind nicht erlaubt.

Die Funktion soll das Argument der Option in `args` (die Definition dieses `struct` finden Sie in `client.h`) speichern.

Im Falle eines fehlerhaften Aufrufs soll das Programm mit einer Usage-Meldung und dem Exit-Code `EXIT_FAILURE` terminieren. Rufen Sie dazu die Funktion `usage( char *msg )` auf.

Hinweise:

- Zum Parsen der Argumente kann die Funktion `getopt(3)` verwendet werden.

1.1 Beispiele

Gültige Aufrufe:

```
$ ./client -p PASSWORD
```

Ungültige Aufrufe:

```
$ ./client -p
  ./client: option requires an argument -- 'p'
  Usage: ./client -p PASSWORD
  invalid option
$ ./client -p PASSWORD SOMETHING ELSE
  Usage: ./client -p PASSWORD
  invalid number of positional arguments
```

2 Gemeinsame Ressourcen öffnen (10)

Vervollständigen Sie die Funktion `void allocate_resources(void)` in `client.c` mit dem Öffnen des gemeinsamen Speichers und der Semaphoren.

Das vorgegebene Server Programm erstellt den gemeinsamen Speicher und die geteilten Semaphore. Ihr Client muss die bereits bestehenden geteilten Resourcen öffnen. In der Datei `common.h` sind Macros für die Namen des geteilten Speichers und der geteilten Semaphore definiert, welche in Ihrem Code verwendet werden müssen.

Öffnen Sie das POSIX Shared Memory Objekt (verwenden Sie das Macro `SHM_NAME` als Namen). Nachdem dem Öffnen soll der gemeinsame Speicher in den Speicher des Prozesses gemapped werden. Der File Descriptor des gemeinsamen Speichers soll in der Variable `shmfd` gespeichert werden und die Addresse des Mappings in der Variable `shmp`. Achten Sie darauf, dass Ihr Prozess sowohl auf den gemeinsame Speicher als auch auf dessen Mapping Lese- und Schreibzugriff hat. Falls kein gemeinsamer Speicher mit diesem Namen existiert, soll Ihr Programm mit einem Fehlerstatus terminieren.

Weiters müssen 3 POSIX Semaphore geöffnet werden (verwenden Sie hierbei die vordefinierten Macros `SEM_NAME_REQUEST`, `SEM_NAME_CLIENT` und `SEM_NAME_RESPONSE` für die Namen der Semaphore). Die Addressen der Semaphor-Objekte sollen in den Variablen `sem_request`, `sem_response` und `sem_client` gespeichert werden, entsprechend den Bezeichnungen der Macros. Falls keine Semaphore mit diesen Namen existieren, soll Ihr Programm mit einem Fehlerstatus terminieren.

Falls es zu einem Fehler kommt, soll das Programm mit dem Exit-Code `EXIT_FAILURE` beendet werden. Sie können dazu die Funktion `error_exit("my err msg")` verwenden.

Hinweise:

- Unter `shm_overview(7)` finden Sie einen Überblick des POSIX Shared Memory und unter `sem_overview(7)` einen Überblick der POSIX Semaphore.

3 Passwort verarbeiten (10)

Vervollständigen Sie die Funktion `void process_password(const char *password, char *hash)` in `client.c`, um darin das Passwort zu verarbeiten.

Schreiben Sie zunächst das Passwort, welches als C-String mittels des Arguments `password` übergeben wird, in den gemeinsamen Speicher. Weisen Sie anschließend den Server mittels des in der Einleitung gezeigten Protokolls an, für dieses Passwort den entsprechenden Hash zu generieren. Sobald der Server anzeigt, dass er mit der Generierung des Hash fertig ist, soll dieser aus dem gemeinsamen Speicher gelesen werden und wiederum als C-String an die Adresse, auf die der Zeiger `hash` verweist, geschrieben werden.

Ergänzen die Funktion um Synchronisierungs-Code, mit dem garantiert wird, dass stets nur ein Client gleichzeitig auf den geteilten Speicher zugreifen kann.

Falls es zu einem Fehler kommt, soll das Programm mit dem Exit-Code `EXIT_FAILURE` beendet werden. Sie können dazu die Funktion `error_exit("my err msg")` verwenden.

Implementierung Testen

Um Ihre Implementierung zu testen, steht Ihnen das Server-Programm `server` zur Verfügung. Dieses Programm akzeptiert keine Optionen oder Argumente.

Synopsis

```
./server
```

Der Server legt die geteilten Resourcen an und wartet anschließend in einer Schleife auf Anfragen von Clients. Sobald er eine Anfrage erhält, wird der Inhalt des geteilten Speichers gelesen, davon ein Hash berechnet, und dieser schließlich wieder in den geteilten Speicher geschrieben.

Verwendungs-Beispiel:

```
$ ./server &
[1] 4025
$ ./client -p password
Hash: 5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
detach shared memory
```

Hinweis: Das `&` am Ende des Server-Aufrufs startet diesen als Hintergrundprozess. Mit dem Befehl `fg` kann ein Hintergrundprozess wieder in den Vordergrund geholt und dann mit `Ctrl+C` beendet werden:

```
$ fg
./server
^C
$
```