UNIVERSITÄT OSNABRÜCK

marcel.luetkedreimann [☺] uni-osnabrueck.de

Übung zur Vorlesung
Betriebssysteme
Wintersemester 2023/24

Programmierblatt 3: Deadlocks

Programmierblatt 3: Deadlocks

Verklemmung auf der Straße

➪ **Abgabe der Lösungen bis Montag, 18. Dezember 16:00 im AsSESS**

Im Folgenden soll eine Straßenkreuzung simuliert werden. Auf den Straßen fahren Käfer, welche gerade über die Kreuzung hinüberfahren wollen. Die Käfer halten sich strikt an die „Rechts vor links"-Regel. Kommen 4 Käfer gleichzeitig an die Kreuzung kann ein Deadlock entstehen, weil alle Käfer auf ihren rechten Nachbarn warten. Das Beispiel wurde in den Vorlesungsfolien „Verklemmungen" auf Folie 5 behandelt.

Geben Sie Ihre Entwicklungsschritte jeweils in den angegebenen Dateien ab.

1 Das Problem an der Kreuzung (35 Punkte)

In der ersten Aufgabe soll die Kreuzung und das Verhalten der Käfer implementiert werden. Im Folgenden sollen Sie schrittweise ein Programm erstellen, das mehrere Käfer simuliert, welche aus einer bestimmten Richtung(Nord, Ost, Süd oder West) über die Kreuzung fahren wollen. Hat ein Käfer die Kreuzung überquert, soll das Programm einen neuen Käfer aus der bestimmten Richtung losfahren lassen. Von diesem Programm wird dann für jede Richtung der Kreuzung eine Instanz gestartet. Ob Käfer an den verschiedenen Positionen der Kreuzung stehen, werden wir durch POSIX-Semaphoren modellieren. Jedes Programm bekommt die Startposition an der Kreuzung und die Position, welche Vorfahrt hat als Shell-Parameter übergeben. Die Kreuzung mit 4 Fahrtrichtungen lässt sich also wie folgt starten:

```
./Cross_a north west &
./Cross_a east north &
./Cross_a south east &
./Cross_a west south &
```

1. Erstellen Sie zunächst die benötigte Anzahl an benannten POSIX-Semaphoren in einem separaten kleinen Hilfsprogramm mit **sem_open(3)**. Überlegen Sie sich, mit welchem Wert die Semaphoren initialisiert werden müssen, um eine Kreuzungszufahrt modellieren zu können. Die hier vergebenen Namen können Sie dann als Shell-Parameter für das Programm nutzen. Für jeden erstellten Semaphor sollte nach dem Aufruf Ihres Programms ein Eintrag mit dem Namen „sem.*SEMAPHORENNAME*" im Verzeichnis `/dev/shm` zu sehen sein.
   Das Hilfsprogramm kann beim Entwickeln Ihrer Lösung auch zum Zurücksetzen der Kreuzung (also der verschiedenen Semaphor-Zähler) verwendet werden, wenn Sie die Semaphoren vor dem Anlegen zunächst mit **sem_unlink(3)** löschen. Dabei sollten Sie Fehler ignorieren, damit das Programm auch läuft, wenn es noch keine Semaphoren gibt.
   Das Hilfsprogramm muss nicht abgegeben werden.
2. In Ihrem Käfer-Programm müssen Sie zunächst die Semaphoren mit den über die Kommandozeile übergebenen Namen öffnen. Lassen Sie danach in einer Endlosschleife das Programm mit **sem_wait(3)** zunächst warten, bis sich kein Käfer mehr auf der Startposition befindet. Anschließend soll der Käfer warten, bis die Vorfahrtposition frei ist. Geben Sie vor und nach dem

sem_wait auf der Standardausgabe jeweils aus, dass der Käfer die entsprechende Position auf der Kreuzung erreichen will bzw. erreicht hat.

3. Zwischen dem Erreichen der eigenen Position und dem Prüfen der Vorfahrt macht der Käfer eine kurze Pause. Nutzen Sie dafür die Funktion **sleep(3)** bzw. **usleep(3)**. (s. Hinweis)
4. Nachdem der Käfer seine Startposition erreicht hat und niemand an der Vorfahrtsposition steht, soll ausgegeben werden, dass der Käfer die Kreuzung befährt. Wenn der Käfer über die Kreuzung fährt, belegt er damit die Vorfahrtsposition, damit es nicht zu einem Unfall kommt. Lassen Sie das Programm nun für eine Sekunde schlafen, um dem Käfer etwas Zeit zu geben die Kreuzung zu überqueren. Anschließend ist die Startposition und Vorfahrtsposition wieder frei. Geben Sie die einzelnen Schritte wieder auf der Standardausgabe aus.
   Bei allen Ausgaben sollten Sie die Prozess-ID des Käfers mit ausgeben, um die verschiedenen Fahrzeuge unterscheiden zu können.
5. Um die Käfer geordnet beenden zu können, bietet es sich an, einen Signalhandler für das Signal **SIGINT** anzumelden, der die Ausführung der Hauptschleife abbricht, wodurch die Semaphoren am Ende wieder ordnungsgemäß freigegeben und geschlossen werden können. Dazu können Sie im Signalhandler ein Flag setzen, das in der Schleife an den entscheidenden Stellen abgefragt wird. Denken Sie daran, alle Semaphoren wieder freizugeben. Verwenden Sie zum Setzen des Signalhandlers *unbedingt* die Funktion **sigaction** und *nicht* signal (siehe Aufgabenteil 3)!
   Mit einem Aufruf von `kill -INT` *`ProzessId`* können sie dann den einzelnen Käfern das Kommando zum Abbrechen schicken.

Eine Ausgabe könnte dann in etwa so aussehen:

```
Kaefer 27840: Ich komme von north.
Kaefer 27840: Ich stehe nun bei north.
Kaefer 27840: Ist west frei?
Kaefer 27840: Es ist frei!
Kaefer 27840: Ich fahre los!
Kaefer 27846: Ich komme von west.
Kaefer 27842: Ich komme von east.
Kaefer 27842: Ich stehe nun bei east.
Kaefer 27840: Ich bin angekommen!
...
```

An irgendeiner Stelle dürften die Programmausgaben abbrechen, weil es zum Deadlock zwischen den Prozessen kommt.

Hinweise:

- Als Vorgabe finden Sie ein Makefile, welches die verschiedenen Aufgabenteile bauen und starten kann. Die Verwendung des Makefile ist mit Kommentaren im Makefile erklärt.
- Um eine zeitliche Synchronisation der Prozesse zu vermeiden, können Sie die Wartezeiten zufällig bestimmen. Für die Wartezeit zwischen **sem_wait(3)** können Sie folgenden Codeschnipsel verwenden: `usleep(10 + 30 * (rand() % 100));` Die Wartezeit zwischen einzelnen Schleifeniterationen sollte größer gewählt werden: `usleep(1000000 - 100 * (rand() % 10));` Um in jeder Käfer-Instanz unterschiedliche Zufallszahlen zu bekommen, müssen Sie am Programmbeginn *einmalig* mit der Funktion **srand(3)** den Pseudozufallsgenerator initialisieren. Für unsere Zwecke reicht etwa der Aufruf `srand(getpid());` aus. Sollten sich im späteren Verlauf keine ´Deadlocks provozieren lassen, können Sie an den Schlafzeiten nachjustieren.
- Wenn sich die Käfer nicht anders beenden lassen, erzwingen Sie jeweils mit `kill ProzessId` bzw. `kill -9 ProzessId` den Abbruch der Käfer und setzten Sie die Semaphoren anschließend mit Ihrem Hilfsprogramm zurück. Das Zurücksetzen der Semaphoren kann bei ungewöhnlichem Programmverhalten recht hilfreich sein.

→ Cross_a.c

2 Die Polizei wird eingesetzt (50 Punkte)

Mittlerweile wurde festgestellt, dass es häufig dazu kommt, dass keiner der Käfer die Kreuzung befahren kann. Deshalb wurde die Polizei gerufen, welche die Situation beobachten soll und erkennt, wenn ein Deadlock vorliegt. Die Polizei soll in regelmäßigen Abständen überprüfen, welche der Käfer blockiert sind. Sind alle Käfer blockiert, meldet die Polizei den Deadlock auf der Standardausgabe.

1. Damit die Polizei mit einem Käfer kommunizieren kann, benötigt Sie zwei Kommunikationskanäle. Ein Kommunikationskanal wird zum Anfragen des Zustands eines Käfers verwendet. Dafür soll ein Signal verwendet werden. Die Polizei bekommt die PIDs aller Käfer als Argument übergeben. In dem anderen Kanal gibt der Käfer seine Antwort. Dieser Kanal soll über eine **FIFO** (benannten Pipe) realisiert werden. Die benötigte FIFO wird bereits mit **mkfifo((1)** in dem vorgegebenen Makefile erstellt.
2. Erstellen Sie ein Polizei-Programm, welches die Verbindung zu der Pipe aufbaut und im 5 Sekundenabstand das **SIGUSR1**-Signal mittels **kill(2)** an alle Käfer sendet. Anschließend wird die Antwort der Käfer über die Pipe gelesen und geprüft, ob ein Deadlock vorliegt. Im Falle eines Deadlocks soll „DEADLOCK“ ausgegeben werden. Beachten Sie, dass das Öffnen von Pipes solange blockiert, bis das lesende *und* das schreibende Ende geöffnet wurden.
3. Erstellen Sie nun eine Signalbehandlung für das Käfer-Programm, das die oben beschriebene Kommunikation zu der Polizei mithilfe der Pipe übernimmt. Beachten Sie, dass das von der Polizei ausgelöste Signal nicht die Systemaufrufe des Käfers abbrechen sollte. Nutzen Sie daher unbedingt **sigaction(2)** mit dem Flag `SA_RESTART`.
4. Um zu überprüfen, ob ein Käfer blockiert ist, reicht es für unsere Anwendung aus, ein globales Flag vom Typ bool zu benutzen. Setzen Sie dieses direkt vor dem Aufruf von **sem_wait(3)** auf `true` und direkt hinter dem Aufruf wieder auf `false`.

Hinweise:

- Es kann passieren, dass die Polizei einen Deadlock feststellt, obwohl kein Deadlock vorhanden ist. Das kann unter anderem der Fall sein, wenn das Signal der Polizei genau nach einem sem_wait-Aufruf, aber noch vor dem Aktualisieren des Flags auftritt. Dieser Fall darf in diesem Übungsblatt ignoriert werden.

Und ungefähr folgende Ausgabe liefern:

```
...
Kaefer 28425: Ist west frei?.
Kaefer 28429: Ich bin angekommen!.
Kaefer 28427: Ich stehe nun bei east.
Kaefer 28431: Es ist frei!
Kaefer 28431: Ich fahre los!
Kaefer 28427: Ist north frei?
Kaefer 28429: Ich komme von south.
Kaefer 28431: Ich bin angekommen!
Kaefer 28429: Ich stehe nun bei south.
Kaefer 28431: Ich komme von west.
Kaefer 28431: Ich stehe nun bei west.
Kaefer 28429: Ist east frei?
Kaefer 28431: Ist south frei?
DEADLOCK
```

→ Cross_b.c | → Police_b.c

3 Die Polizei regelt den Verkehr (15 Punkte)

In diesem Aufgabenteil soll implementiert werden, dass die Polizei den Verkehr regelt, wenn ein Deadlock festgestellt wurde.

Gemäß dem POSIX-Standard werden die meisten Systemaufrufe durch Signal-Handler unterbrochen. Das gilt auch für **sem_wait(3)**. So unterbrochene Systemaufrufe liefern dann einen Fehler und `errno` wird auf `EINTR` gesetzt. Dieses Verhalten wollen wir ausnutzen, um einen Deadlock aufzulösen.

1. Wenn die Polizei einen Deadlock festgestellt hat, sollen die Käfer neu gestartet werden. Dafür wird mithilfe von **kill(2)** das Signal **SIGALRM** an den Käfer-Prozess gesendet. Erstellen Sie für dieses Signal einen Signal-Handler im Käfer und melden diesen an. Überlegen Sie, wie viel dieser Handler angesichts des oben beschriebenen Verhaltens unterbrochener Systemaufrufe tun muss und welche Flags für die Anmeldung gesetzt werden müssen.
2. Werten Sie den Fehlercode von sem_wait-Aufrufen aus und geben Sie gegebenenfalls eine Position der Kreuzung wieder frei, wenn der Systemaufruf durch ihren Signal-Handler unterbrochen wurde. Das Käfer-Programm soll danach wieder am Anfang der Hauptschleife weitermachen.

Die Ausgabe Ihres Programms sollte nun in etwa so aussehen:

```
...
Kaefer 33699: Ich stehe nun bei east.
Kaefer 33697: Ich stehe nun bei north.
Kaefer 33699: Ist north frei?
Kaefer 33697: Ist west frei?
DEADLOCK!
Kaefer 33697: Ich habe keine Gedult mehr zu warten. Ich kehre um!
Kaefer 33697: Ich komme von north.
Kaefer 33699: Ich habe keine Gedult mehr zu warten. Ich kehre um!
Kaefer 33697: Ich stehe nun bei north.
...
```

→ Cross_c.c | → Police_c.c

Tipps zu den Programmieraufgaben:

- Kommentieren Sie Ihren Quellcode ausführlich, so dass wir auch bei Programmierfehlern im Zweifelsfall noch Punkte vergeben können!
- Denken Sie daran, dass viele Systemaufrufe fehlschlagen können! Fangen Sie diese Fehlerfälle ab (die Aufrufe melden dies über bestimmte Rückgabewerte, siehe die jeweiligen man-Pages), geben Sie geeignete Fehlermeldungen aus (z.B. unter Zuhilfenahme von **perror(3)**) und beenden Ihr Programm danach ordnungsgemäß.
- Die Programme sollen dem C11- und POSIX-Standard entsprechen sich mit dem gcc auf aktuellen Linux-Rechnern wie denen im CIP-Pool oder der BSVM übersetzen lassen. Z.B.:
  `gcc -std=c11 -Wall -D_GNU_SOURCE -o Cross_x Cross_x.c`
  Weitere (nicht zwingend zu verwendende) nützliche Compilerflags sind: `-Wextra -Wpedantic -Werror -D_POSIX_SOURCE`
- Achten Sie darauf, dass sich der Programmcode ohne Warnungen übersetzen lässt, z.B. durch Nutzung von `-Werror`.