UNIVERSITÄT OSNABRÜCK

marcel.koeppen [☺] uni-osnabrueck.de
marcel.luetkedreimann [☺] uni-osnabrueck.de

Übung zur Vorlesung
Betriebssysteme
Wintersemester 2021/22

Programmierblatt 3: Deadlocks
Philosophers Diner: Bring your own forks

➪ **Abgabe der Lösungen bis Montag, 13. Dezember 14:00 im AsSESS**

Geben Sie Ihre Entwicklungsschritte jeweils in den angegebenen Dateien ab.

1 Die Philosophen haben Hunger (6 Punkte)

In der ersten Aufgabe soll das bekannte Philosophenproblem (s. dazu Philosophenproblem (Wikipedia)) nachgebaut werden. Bei uns sollen aber beliebig viele Philosophen am Tisch sitzen können. Im Folgenden sollen Sie schrittweise ein Programm erstellen, das einen einzelnen Philosophen implementiert. Von diesem Programm wird dann für jeden Philosophen am Tisch eine Instanz gestartet. Die benötigten Gabeln werden wir durch benannte POSIX-Semaphoren modellieren und jedem Philosophen das passende Gabel-Paar als Shell-Parameter übergeben.

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 Gabel modellieren zu können. Die hier vergebenen Namen können Sie dann als Shell-Parameter für das Philosophen-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 Gabeln (also der 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 Philosophen-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 auf die linke Gabel und danach auf die rechte Gabel warten. Geben Sie vor und nach dem sem_wait auf der Standardausgabe jeweils aus, dass der Philosoph die entsprechende Gabel nehmen will bzw. die Gabel genommen hat.
3. Zwischen dem Aufnehmen der linken und der rechten Gabel macht der Philosoph eine kurze Pause. Nutzen Sie dafür die Funktion **sleep(3)** bzw. **usleep(3)**.
4. Nachdem der Philosoph beide Gabeln erhalten hat, soll ausgegeben werden, dass der Philosoph isst. Lassen Sie das Programm nun für eine Sekunde schlafen, um dem Philosophen Zeit zum Essen zu geben. Anschließend legt der Philosoph die Gabeln wieder weg und wartet ein letztes Mal, um zu denken. Geben Sie die einzelnen Schritte wieder auf der Standardausgabe aus.
   Bei allen Ausgaben sollten Sie die Prozess-ID des Philosophen mit ausgeben, um die verschiedenen Philosophen unterscheiden zu können.

Beim Aufruf eines einzelnen Philosophen mit

```
./philosoph gabel1 gabel2
```

sollte die Ausgabe dann in etwa so aussehen:

```
Philosoph 83825 will die linke Gabel nehmen.
Philosoph 83825 hat die linke Gabel.
Philosoph 83825 will die rechte Gabel nehmen.
Philosoph 83825 hat die rechte Gabel.
Philosoph 83825 isst.
Philosoph 83825 legt die linke Gabel weg.
Philosoph 83825 legt die rechte Gabel weg.
Philosoph 83825 denkt.
Philosoph 83825 will die linke Gabel nehmen.
Philosoph 83825 hat die linke Gabel.
...
```

Nun können Sie mehrere Philosophen gemeinsam essen lassen. Der entsprechende Aufruf für drei Philosophen könnte so aussehen:

```
./philosoph gabel3 gabel1 &
./philosoph gabel1 gabel2 &
./philosoph gabel2 gabel3 &
```

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

Hinweise:

- 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 Philosophen-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.
- Um die Philosophen 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 Gabeln 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 Philosophen das Kommando zum Abbrechen schicken.
- Wenn sich die Philosophen nicht anders beenden lassen, erzwingen Sie jeweils mit `kill ProzessId` bzw. `kill -9 ProzessId` den Abbruch der Philosophen und setzten Sie die Semaphoren anschließend mit Ihrem Hilfsprogramm zurück. Das Zurücksetzen der Semaphoren kann bei ungewöhnlichem Programmverhalten recht hilfreich sein.

→ `philosoph_a1.c`

2 Die Philosophen lernen (8 Punkte)

Mittlerweile haben die Philosophen festgestellt, dass es häufig dazu kommt, dass keiner von ihnen essen kann. Deshalb haben Sie sich ein System überlegt, mit dem Sie erkennen können wenn ein Deadlock vorliegt. Vor Beginn der Tischrunde wird ein Philosoph festgelegt, der dafür zuständig ist.

Dieser leitende Philosoph soll dann in regelmäßigen Abständen überprüfen, ob er selbst blockiert ist. Ist das der Fall, wartet er eine gewisse Zeitspanne und überprüft sich erneut. Ist er dann immer noch blockiert, teilt er dies seinem linken Tischnachbar mit. Dieser überprüft dann auch, ob er blockiert ist. Sollte das der Fall sein, leitet er die Nachricht unverändert an seinen linken Nachbarn weiter. Falls nicht, teilt er seinem linken Nachbarn mit, dass mindestens ein Philosoph nicht blockiert ist, so dass alle nachfolgenden Philosophen diese Nachricht ohne weitere Selbstprüfung weiterleiten können. Das wird solange wiederholt, bis die Nachricht wieder beim leitenden Philosophen ankommt. Sollte dieser so feststellen, dass alle Philosophen blockiert sind, meldet er einen Deadlock.

1. Übergeben Sie einem der Philosophen beim Start einen zusätzlich Kommandozeilenparameter, der ihn als leitenden Philosophen markiert. Werten Sie diesen Parameter in ihrem Programm entsprechend aus.
2. Damit die Prozesse miteinander kommunizieren können, benötigen Sie einen Kommunikationskanal zu ihren Nachbarn. Dieser soll über eine passende Anzahl von **FIFOs** (benannten Pipes) realisiert werden. Zum Erstellen der benötigten FIFOs mit **mkfifo((1)** bietet sich ein kleines Shell-Skript oder Makefile an. Die hierbei vergebenen Namen können dann wiederum als Shell-Parameter an die Philosophen übergeben werden.
3. Mit den erstellten Pipes soll dann ein Ring erzeugt werden: `Philosoph_1 (write)`$\Longrightarrow$`(read) Philosoph_n (w)`$\Longrightarrow$`(r) Philosoph_n-1 (w)`$\Longrightarrow$ `...` $\Longrightarrow$`(r) Philosoph_2 (w)`$\Longrightarrow$`(r) Philosoph_1`.
   Da das Öffnen von Pipes solange blockiert, bis das lesende *und* das schreibende Ende geöffnet wurde, müssen Sie sich überlegen, wie Sie beim Öffnen der Pipes keinen Deadlock verursachen.
4. Erstellen Sie nun einen neuen Thread, der die oben beschriebene Kommunikation zwischen den Philosophen übernimmt. Es reicht aus, ein einzelnes Zeichen über den Ring zu schicken, das angibt, ob bisher alle Philosophen blockiert sind. Kommt beim leitenden Philosophen die Nachricht an, dass alle Philosophen auf dem Ring blockiert sind, soll er „DEADLOCK“ ausgeben.
5. Um zu überprüfen, ob ein Philosoph blockiert ist, reicht es für unsere Anwendung aus, ein globales Flag vom Typ bool zu benutzen. Setzen Sie dieses in der Implementierung des Essverhaltens ihres Philosophen direkt vor dem Aufruf von **sem_wait(3)** auf `true` und direkt hinter dem Aufruf wieder auf `false`. Das Flag kann dann vom Kommunikationsthread regelmäßig, z.B. alle 5 Sekunden, abgefragt werden.

Ein Aufruf für drei Philosophen könnte jetzt etwa so aussehen:

```
./philosoph gabel3 gabel1 pipe3 pipe1 istboss
./philosoph gabel1 gabel2 pipe1 pipe2
./philosoph gabel2 gabel3 pipe2 pipe3
```

Und ungefähr folgende Ausgabe liefern:

```
...
Philosoph 257061 legt die rechte Gabel weg.
Philosoph 257061 denkt.
Philosoph 257063 hat die linke Gabel.
Philosoph 257061 will die linke Gabel nehmen.
Philosoph 257068 will die rechte Gabel nehmen.
Philosoph 257061 hat die linke Gabel.
Philosoph 257061 will die rechte Gabel nehmen.
Philosoph 257063 will die rechte Gabel nehmen.
DEADLOCK
```

→ `philosoph_a2.c`

3 Die Philosophen sind kooperativ (3 Punkte)

In diesem Aufgabenteil soll implementiert werden, dass der leitende Philosoph seine Gabel freigibt, wenn er einen Deadlock feststellt.

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. An der Stelle, an der Sie bisher „DEADLOCK“ ausgegeben haben, soll nun mithilfe von **pthread_kill(3)** das Signal **SIGALRM** an den Haupt-Thread gesendet werden. Erstellen Sie für dieses Signal einen Signal-Handler und melden diesen an. Überlegen Sie, wie viel dieser Handler angesichts des oben beschriebenen Verhaltens unterbrochener Systemaufrufe tun muss. **Hinweis**: Um die ID eines Threads herauszufinden, können Sie aus dem fraglichen Thread heraus **phread_self(3)** aufrufen.
2. Werten Sie den Fehlercode von sem_wait-Aufrufen aus und geben Sie gegebenenfalls die erste Gabel wieder frei, wenn der Systemaufruf durch ihren Signal-Handler unterbrochen wurde. Der Philosoph soll danach wieder am Anfang seiner Hauptschleife weitermachen.

ACHTUNG: Signal-Handler die mit **signal(2)** angemeldet werden, unterbrechen zwar Systemaufrufe, starten diese aber automatisch neu! Nutzen Sie daher unbedingt **sigaction(2)** ohne das Flag `SA_RESTART`.

Die Ausgabe Ihres Programms sollte nun in etwa so aussehen:

```
...
Philosoph 265801 will die rechte Gabel nehmen.
Philosoph 265806 will die rechte Gabel nehmen.
Philosoph 265799 will die rechte Gabel nehmen.
Philosoph 265803 will die rechte Gabel nehmen.
DEADLOCK
Philosoph 265799 wartet nicht länger und legt die linke Gabel weg.
Philosoph 265806 hat die rechte Gabel.
Philosoph 265806 isst.
Philosoph 265799 will die linke Gabel nehmen.
...
```

→ `philosoph_a3.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 -pthread -o philosoph philosoph_ax.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`.