Technische Universität Berlin
Fachgebiet
Distributed and Operating Systems

Sommersemester 2025

Aufgabenblatt 3

zu – Systemprogrammierung –

Prof. Dr. Odej Kao

Aufgabe 3.1: Synchronisation (Tafelübung)

Die Abläufe zwischen Verkäufern und Kunden in einem Dönerladen sollen synchronisiert werden. In diesem Dönerladen gibt es einen Spieß und mindestens zwei Verkäufer. Der Spieß kann nur von einem Verkäufer gleichzeitig genutzt werden. Auf den Salat kann von allen gleichzeitig zugegriffen werden.

a) Im Folgenden ist der Verkäufer-Prozess in Pseudocode beschrieben. Dabei machen wir uns erst einmal noch keine Sorgen um zu viel produzierte Döner. Ergänzen Sie die nötige Synchronisation.

```
Variablen:
int döner = 0;

Verkäufer:
while(true){
  fleischSchneiden();
  salatUndSauce();
  döner++;
}
```

b) Kunden betreten in unvorhersagbaren Abständen den Laden, um einen Döner zu kaufen (wir simulieren das durch startende Kunden-Prozesse). Sie können nur Döner essen, wenn auch Döner fertig sind, andernfalls müssen sie warten. Erweitern Sie Ihre Lösung aus der letzten Aufgabe dafür um den nachfolgenden angegebenen Kunden-Prozess und die notwendige Synchronisation.

```
Kunde:
  döner--;
  dönerEssen();
```

c) Verkäufer sollen nur dann etwas produzieren, wenn auch ein Kunde auf den Döner wartet. Ergänzen Sie Ihre Lösung aus der letzten Aufgabe um die dafür notwendige Synchronisation.

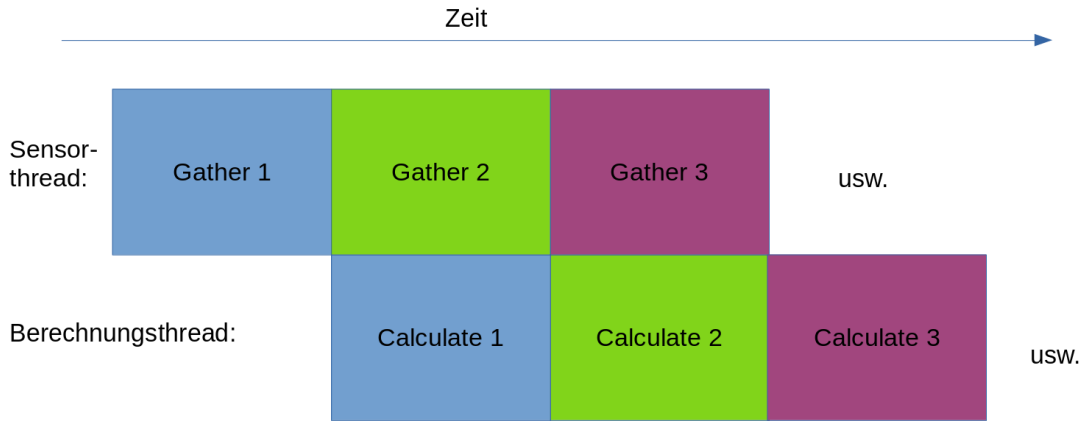
Aufgabe 3.2: Synchronisation/Kooperation (Tafelübung)

Sie sollen einen Smart-Home-Wettersensor entwickeln. Dieser soll mithilfe von eingebauten Sensoren Wetterdaten erfassen und diese danach auswerten.
Die Funktion `gather_data(char *buffer)` wird zur Datenmessung aufgerufen, und schreibt während des Messens Messdaten in den übergebenen Speicher.
Die Funktion `calculate_forecast(char *buffer)` wertet die übergebenen Messdaten aus und schickt das Ergebnis an gewünschte Geräte im Heimnetzwerk.

Der Sensor misst die Umgebungsdaten mithilfe von `gather_data` (dies dauert ca. 30 min). Nach jedem neuen Datensatz soll eine neue Wettervorhersage berechnet werden (`calculate_forecast`). Das

Berechnen der Vorhersage dauert ebenfalls ca. 30 min. Während die alten Messdaten ausgewertet werden, sollen die Sensoren direkt wieder mit dem Messen von neuen Werten beginnen.

Hinweis:
Ein beispielhafter zeitlicher Ablauf
(die Nummern stehen hier für beispielhafte Datensätze):



Global:

```
char buffer[1024];
state s = gather;        // Werte: 'gather' oder 'copy_data'
```

Sensor-Thread:

```
while(1) {
        gatherData(&buffer);
        s = copy_data;
}
```

Berechnungs-Thread:

```
while(1) {
        char internal_buffer[1024];
        memcpy(internal_buffer,buffer,1024);
        s = gather;
        calculate_forecast(internal_buffer);
}
```

Hinweis: Gehen Sie in dieser Aufgabe davon aus, dass Signale, auf die zum Zeitpunkt des Sendens nicht gewartet wird, gespeichert werden.

a) Die beiden Threads sollen nun nebenläufig ausgeführt werden. Nennen Sie ein praktisches Problem, das dabei auftreten kann.

b) Was sind *Spurious Wakeups*? Wie kann man sicherstellen, dass die eigentliche Bedingung erfüllt ist, auch im Fall von *Spurious Wakeups*?

c) Verbessern Sie obiges Programm mit *signal/wait* und *mutex*, sodass Probleme verhindert werden. Da die Datenerhebung(`Gather_Data`) sehr lange dauert, sollte damit schon begonnen werden, während der alte Datensatz ausgewertet wird(`calculate_forecast`)(siehe Beispielablauf). Außerdem sollten beim Auswerten keine Datensätze übersprungen werden, um stets aktuelle Daten zu gewährleisten. Ihre Lösung sollte auch Spurious Wakeups berücksichtigen.
Hinweis: Sie können zusätzliche globale Variablen verwenden.

d) Welche Arten der expliziten Prozess- /Threadinteraktion gibt es? Um welche Art der Interaktion handelt es sich hier?

Aufgabe 3.3: Atomare Operationen und Races (Tafelübung)

a) Was sind atomare Operationen und wie hängen sie mit Data Races zusammen?

b) Es werden folgende 2 Threads in einem Ein-Prozessor-System gestartet mit x = y = 0. Teilen sie die Operationen in ihre Bestandteile auf. Was sind mögliche Werte von x und y nach der Ausführung? Welche Ausgaben auf stdout sind möglich ?

Listing 1: Thread A

```
1  void *thread_A(int* x, int* y) {
2    *y += 1;
3    *x += 5;
4  }
```

Listing 2: Thread B

```
1  void *thread_B(int *x, int* y) {
2    *y += 3;
3    if(*y > *x){
4      printf("%i>%i", *y, *x);
5    }
6  }
```

c) angenommen alle Operationen in den oberen Threads sind atomar. Ist die Ausgabe immer korrekt ?

Aufgabe 3.4: UNIX Commands I (Tafelübung)

a) Nennen Sie die Unterschiede zwischen dem **Bash-Terminal** (unter Linux/macOS) und dem **Windows-Terminal** (CMD oder PowerShell). Notieren Sie kurz:

- Herkunft und typische Verwendung,
- Unterschiede in der Kommandosprache und Funktionalität,
- wie Bash unter Windows nutzbar ist.

b) Nutzen Sie den Bash-Terminal[1], um die gegebenen Bücher herunterzuladen und in der Directory `sacred-books` abzulegen. Nutzen Sie hierfür die folgenden Links:

[1] `https://linuxstans.com/bash-cheat-sheet`

- `https://www.gutenberg.org/cache/epub/1513/pg1513.txt`
- `https://www.gutenberg.org/cache/epub/1342/pg1342.txt`

c) Sie versuchen nicht-priviligierte Nutzer davon abzuhalten, die Inhalte der Dateien zu ändern, indem Sie

- die Schreibrechte vom Verzeichnis entfernen,
- die Schreibrechte von allen Dateien in dem Verzeichnis entfernen.

d) Was ist der Unterschied zwischen den beiden Ansätzen?

e) Probieren Sie im Terminal die Befehle `ls` und `man ls` aus. Notieren Sie, was die Befehle bewirken und wie Sie mithilfe von `man` weitere Informationen zu anderen Befehlen erhalten können.

Aufgabe 3.5: UNIX Commands II (Selbststudium)

a) Recherchieren Sie eigenständig, welche verschiedenen Terminal-Shells unter Linux existieren können (z. B. bash, zsh, fish, sh). Nennen Sie mindestens drei verschiedene Shells und beschreiben Sie kurz ihre Besonderheiten oder typische Einsatzgebiete.

b) Nutzen Sie den Bash-Terminal, um ein kleines vorgefertigtes Programm namens `hello-world` herunterzuladen. Speichern Sie es zunächst in Ihrem Home-Verzeichnis.

Download-Link:
`https://raw.githubusercontent.com/dos-group/sysprog/refs/heads/main/bin/hello-world`

c) Legen Sie ein neues Verzeichnis `/opt/bin` an (sofern es nicht existiert) und verschieben Sie die heruntergeladene Datei dorthin.
Ändern Sie anschließend:

- die Dateiberechtigungen so, dass alle Nutzer das Programm ausführen können,
- den Besitzer der Datei zu `root`.

d) Prüfen Sie, ob sich das Programm bereits im `PATH` befindet. Falls nicht:

- Erstellen Sie einen symbolischen Link nach `/usr/local/bin/hello-world`, sodass das Programm systemweit verfügbar ist.
- Führen Sie das Programm erneut nur mit `hello-world` (ohne Pfadangabe) aus.

e) Erstellen Sie eine Bash-Datei (z. B. `install-hello-world.sh`), die alle oben genannten Schritte automatisch durchführt.
Achten Sie darauf, dass die Datei ausführbar ist und alle notwendigen `sudo`-Befehle enthält, um das Skript auf einem typischen UNIX-System erfolgreich auszuführen.

Aufgabe 3.6: POSIX Threads (Selbststudium)

a) Was ist der Unterschied von Threads und Prozessen? Wie sieht dieser im Hinblick auf die POSIX-Bibliothek pThreads aus? Geben Sie zudem Möglichkeiten an, wie Threads untereinander kommunizieren können, sowie berechnete Ergebnisse an den Parent weitergeben, bzw. von diesem bei Start bekommen können.

b) Es ist das folgende Ping/Pong-Programm gegeben. Dieses soll mit der pThreads POSIX-Bibliothek so implementiert werden, dass die Threads im Wechsel „Ping“ und „Pong“ ausgeben. Spurious Wakeups sollen berücksichtigt werden.

Listing 3: main

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  enum action{PING, PONG};
5
6  int main() {
7    enum action *nextAction = malloc(sizeof(enum action));
8    *nextAction = PING;
9
10   thread_ping(nextAction);
11   thread_pong(nextAction);
12
13   free(nextAction);
14 }
```

Listing 4: Thread 1

```
1  void *thread_ping(void *nAction) {
2    enum action *nextAction = (enum action *) nAction;
3
4    while(1) {
5      printf("Ping\n");
6      *nextAction = PONG;
7    }
8  }
```

Listing 5: Thread 2

```
1  void *thread_pong(void *nAction) {
2    enum action *nextAction = (enum action *) nAction;
3
4    while(1) {
5      printf("Pong\n");
6      *nextAction = PING;
7    }
8  }
```