Technische Universität Berlin Fachgebiet Komplexe und Verteilte IT-Systeme Sommersemester 2020	**Aufgabenblatt 4** zu – Systemprogrammierung – Prof. Dr. Odej Kao
Abgabetermin:	**28.06.2020 23:55 Uhr**

Aufgabe 4.1: 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 4.2: POSIX Threads (Tafelübung)

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 1: main

```c
#include <stdio.h>
#include <stdlib.h>

enum action{PING, PONG};

int main() {
  enum action *nextAction = malloc(sizeof(enum action));
  *nextAction = PING;

  thread_ping(&nextAction);
  thread_pong(&nextAction);

  free(nextAction); // not really necessary
}
```

Listing 2: Thread 1

```c
void *thread_ping(void *nAction) {
  enum action *nextAction = (enum action *) nAction;

  while(1) {
    printf("Ping\n");
    *nextAction = PONG;
  }
}
```

Listing 3: Thread 2

```c
void *thread_pong(void *nAction) {
  enum action *nextAction = (enum action *) nAction;

  while(1) {
    printf("Pong\n");
    *nextAction = PING;
  }
}
```

Aufgabe 4.3: Priority Inversion (Tafelübung)

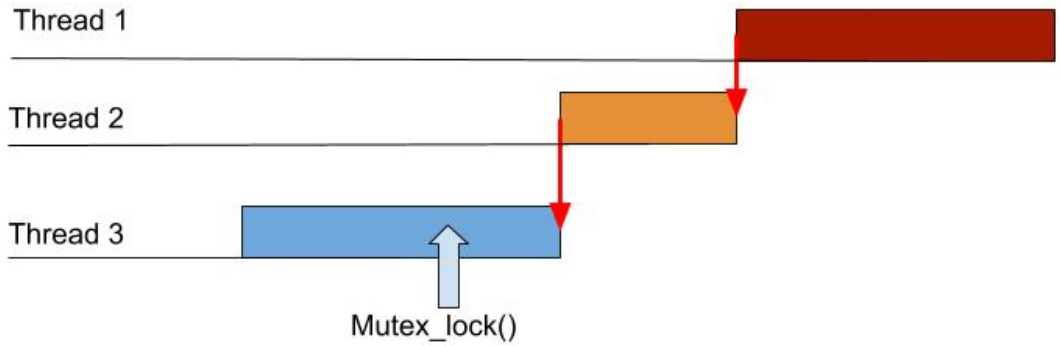
In 1997 ist der Mars Pathfinder auf dem Mars gelandet.
Der hatte einen gemeinsamnen Informationsbus und einen watchdog timer der überprüft, ob das System noch arbeitet. Aufgaben wurden auf Threads verteilt mit verschiedene Prioritäten. Als scheduling algorithmus wurde eine Priority-queue mit Verdrängung benutzt.
Drei Threads waren Periodisch (in Reinfolge von Prioritäten):

Thread 1. Informationsbus-Thread: sehr häufig und hohe Priorität

Thread 2. Kommunikations-Thread, mittel häufig und mittel hohe Priorität (kann sehr lange laufen)

Thread 3. Wetter Daten sammeln: nicht häufig, niedrige Priorität (Braucht wie Thread 1 und 2 auch den Speicher)

a) Nach paar Tagen hat der Watchdog Timer immer des System neu gestartet wegen eines Problems. Überlegen sie was passiert, wenn Thread 3 dran kommt, den geteilten Speicher mit Thread 1 sperrt, dann will Thread 2 den langen Kommunikationsprozess anfangen, und danach will Thread 1 starten.



b) Überlege Sie wie in diesen Fall Thread 3 den Speicher wieder freischalten könte.

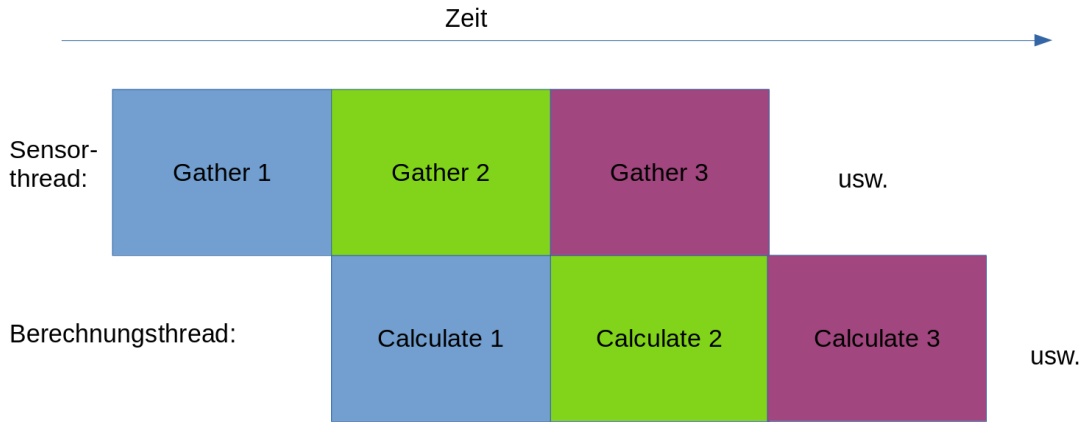
Aufgabe 4.4: Synchronisation/Kooperation (2,6 Punkte) (Theorie)

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 Signal, 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.
(0,2 Punkte)

b) Was sind *Spurious Wakeups*? Und wie kann man sicher stellen, dass die eigentliche Bedingung erfüllt ist, auch in fall von Spurious wakeup?
(0,4 Punkte)

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.
(1.6 Punkte)

d) Welche Arten der expliziten Prozess- /Threadinteraktion gibt es? Um welche Art der Interaktion handelt es sich hier?
(0,2 Punkte)

Aufgabe 4.5: Periodische Prozesse (1,4 Punkte) (Theorie)

Die Firma „Pen&Pencil“ möchte einen neuartigen Stift auf den Markt bringen. Dieser soll speziell in Meetings eingesetzt werden können und folgende Funktionen bieten: A) Die Beschleunigung aufzuzeichen, sodass Geschriebenes einfach digitalisiert werden kann, B) Diese Daten (aus einem Puffer) auf die enthaltene MicroSD-Karte zu schreiben und C) Geschriebenes sofort ohne Verzögerung auf entsprechenden Boards über eine drahtlose Verbindung übertragen. Hierbei ist es wichtig, dass diese Aufgaben ohne Verzögerung möglichst schnell (ohne Verletzung der Deadline) und zuverlässig ausgeführt werden. In der nachfolgenden Tabelle sind die beispielhaften Eckdaten einer solchen Benutzung dargestellt: Dauer der Aufgabe und Periode, die zeitgleich auch die Frist (Deadline) ist. Alle Prozesse starten zeitgleich bei $t = 0$.

Tabelle 1: Prozesse

Prozesse	**Dauer (D)**	**Periode (P)**
A	1	3
B	1	5
C	1	5

a) Existiert für diese Prozesse ein zulässiger Schedule? Wird das notwendige Kriterium erfüllt? **(0,2 Punkte)**

b) Wie könnte dieser aussehen? Geben Sie etwaige Leerzeiten an und markieren Sie die Hyperperiode. **(0,6 Punkte)**

c) Ist Rate-Monotonic-Scheduling (RMS) ein gültiger Schedule? Begründen Sie Ihre Antwort. **(0,2 Punkte)**

d) Was passiert, wenn zu den Prozessen ein weiterer Prozess, Prozess D mit (D=2, P=9), hinzugefügt wird? **(0,4 Punkte)**

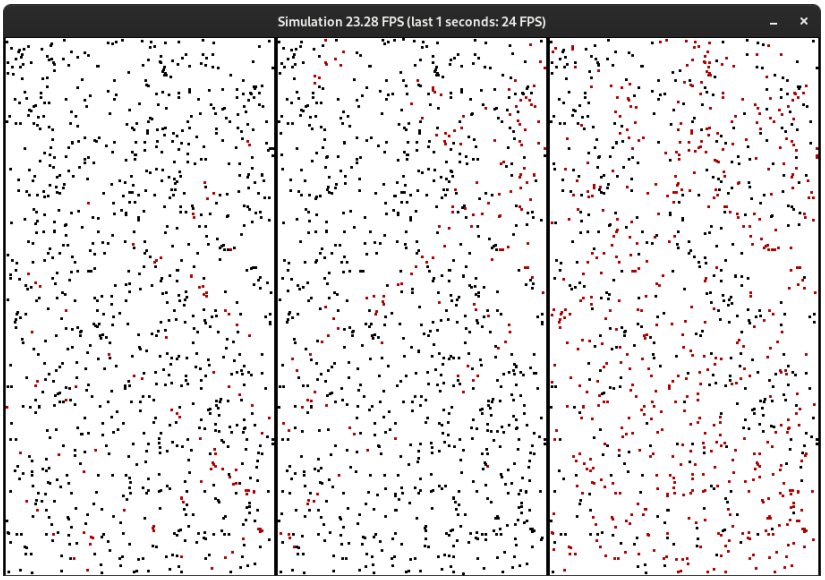
Aufgabe 4.6: Social Distancing Simulator (5 Punkte) (Praxis)

Sie wurden vom Rober Bäcker-Institut beauftragt, die Simulation eines Virusausbruches zu programmieren. In der Simulation soll die Auswirkung von “Social Distancing“ auf die Verbreitungsgeschwindigkeit des Virus untersucht werden. Dazu haben Sie ein Programm entwickelt, welches mehrere Szenarien simuliert. In jedem Szenario werden unterschiedliche Einhaltungsquoten von Social Distancing simuliert. Social Distancing wird dadurch simuliert, dass Personen sich nicht bewegen. Personen infizieren sich, sobald sie sich berühren. Folglich sollte weniger Bewegung zu weniger Infektionen führen. Ihr erster Prototyp funkioniert zwar vollständig, aber die Performance lässt noch zu wünschen übrig. Um die Simulationperformance zu verbessern, sollen Sie in dieser Aufgabe jedes Szenario in einem eigenen Thread laufen lassen.

In der Vorgabe finden Sie die Datei **src/Sim.c**, aus welcher das Programm **Sim** kompiliert wird. Dieses Programm ist der erste funktionierender Prototyp, indem die Szenarien aber noch nicht nebenläufig arbeiten. Die Datei **src/SimThreaded.c** kompiliert zu dem Programm **SimThreaded**, in dem die Szenarien nebenläufig jeweils in einem eigenen Threads laufen sollen. Beide Programme können mit dem Flag *-h* aufgerufen werden, um die optionalen Parameter anzuzeigen, die beim Programmaufruf mitgegeben werden

können. Um diese Aufgabe zu lösen, bearbeiten Sie die Dateien **include/Scenario.h**, **src/Scenario.c** und **src/SimThreaded.c**. Fügen Sie Ihren Code **nur** zwischen *TODO BEGIN* und *TODO END* ein!

Das Programm gibt für jedes Szenario die Anzahl der erkrankten Personen pro Simulationsschritt in einer CSV-Datei aus. Die CSV-Datei können Sie mithilfe eines Tabellenkalkulationsprogrammes (zB. *Microsoft Excel, Libre Calc, Google Sheets*) grafisch darstellen. Das Programm hat auch einen optionalen GUI-Modus. Ein Screenshot des GUI-Modus ist unten abgebildet. Um diesen Modus zu benutzen, folgen Sie den Anweisungen in der **README_GUI.md**. Dieser Modus nicht über SSH ausführbar!



Die Aufgabe ist so konzipiert, dass bei korrekter Implementierung die CSV-Dateien von **Sim** und **SimThreaded** bei gleichen Argumenten identisch sind. Nutzen Sie dies, um die Korrektheit Ihres Programms zu verifizieren!

Wir empfehlen Ihnen die folgende Bearbeitungreihensfolge:

(A) Erweitern Sie die Datei **src/SimThreaded.c**, sodass die Funktion *Sim_Init()* die Szenario Threads startet und *Sim_Cleanup()* auf das selbstständige Beenden der Threads wartet. Speichern Sie sich dazu die Referenz auf die Threads in dem Struct *Sim* ab, indem Sie das Struct erweitern.

(B) Implementieren Sie nun das Grundgerüst der Funktion *Scenario_Main()*. Nutzen Sie zur Koordinierung mit dem Main-Thread die geteilte Adresse der Statusvariable *state*.

(C) Bereiten Sie sich vor, den Zugriff auf den geteilten Speicher abzusichern. Nutzen Sie dazu nur Mutexe und Signale aus der PThread Bibliothek! Erweitern Sie das Struct *Sim* in der Datei **src/SimThreaded.c** um Mutexe und Signale und das Struct *Args* in der Datei **include/Scenario.h** um Pointer zu den Mutexen und Signalen. Behandeln Sie die erstellten Mutexe und Signale in der Funktion *Sim_Init()* und *Sim_Cleanup()*.

(D) Füllen Sie nun das erstellte Grundgerüst in *Scenario_Main()* aus. Implementieren Sie danach die Funktion *Sim_CalcActiveIteration()*. Erweitern Sie auch die Funktion *Sim_End()*, sodass diese die Threads informiert, sich zu terminieren.

Nun sollte Ihr Programm (hoffentlich) funktionieren. Testen Sie Ihr Programm dennoch ausgiebig auf Memory Leaks, Speicherzugriffsfehler und Synchronisationsfehler! Führen Sie diese Tests auch per SSH-Verbindung auf den Uniservern aus! Probieren Sie dabei verschiedene Konfigurationen der Parameter in

der Datei **include/Params.h** aus! Fangen Sie mit einem Szenario-Thread an und erhöhen Sie die Anzahl, sobald die momentane Anzahl stabil läuft. Das Programm **SimThreaded** sollte dabei wesentlich schneller sein als das Vorgabeprogram **Sim**!

Abgabe:

Wie auf der Kursseite beschrieben sollen Sie ihre Lösung in einer ZIP-Datei abgeben. Die ZIP-Datei soll nur ihren modifizierten **src/**- und **include/**-Ordner, das Makefile aus der Vorgabe, den originalen **data/**-Ordner, einen leeren **obj/**-Ordner und eine Textdatei mit den Namen der Gruppenmitglieder beinhalten.

Hinweise:

- Verschaffen Sie sich einen Überlick über die Abläufe aller Threads
- Überlegen Sie, wie Sie die Szenario-Threads mit dem Main-Thread synchronisieren können!
- Berücksichtigen Sie auch Spurious Wakeups
- Nutzen Sie das **Makefile** in den Vorgaben zum kompilieren
- Nutzen Sie *valgrind - -tool=helgrind ./SimThreaded*, um Synchronisationsprobleme zu erkennen. Für weitere Informationen zu helgrind besuchen Sie die folgende Seite: `http://valgrind.org/docs/manual/hg-manual.html`
- Bei einem Deadlock wird die Abgabe mit 0 Punkten bewertet.
- Testen Sie Ihre Implementierung auf Memory Leaks! Nur der nichtgraphische Modus muss Memory Leak frei sein!
- Denken Sie auch daran die erstellten Mutexe und Condition-Variablen wieder zu zerstören mit den Aufrufen *pthread_mutex_destroy(pthread_mutex_t* name_des_Mutex), pthread_cond_destroy(pthread_cond_t* Name_des_Signals)*