Technische Universität Berlin Fachgebiet Komplexe und Verteilte IT-Systeme Sommersemester 2019	**Aufgabenblatt 4** zu – Systemprogrammierung – Prof. Dr. Odej Kao
Abgabetermin:	[1] – 23.06.2019 23:55 Uhr [2] – 23.06.2019 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)

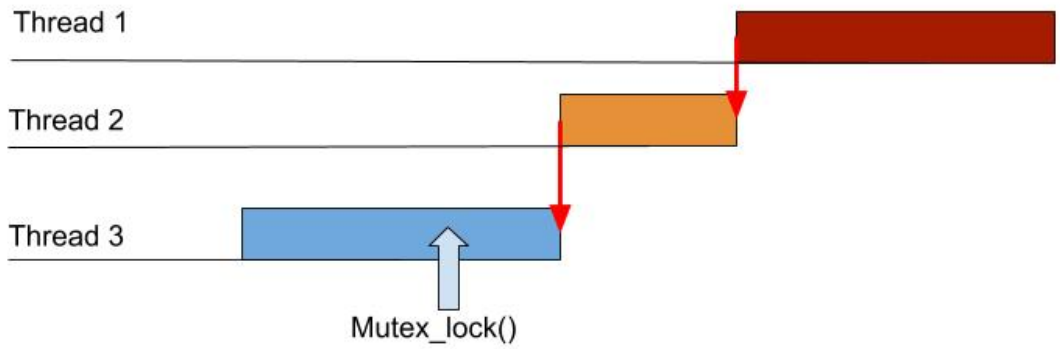
In 1997 ist der Mars Pathfinder auf dem Mars gelandet.
Der hatte einen gemeinsamnen Informationsbus und einen watchdog timer der überprüft, ob das System noch arbeitet. Aufgaben wurden auf Threads verteilt mit verschiedene Prioritäten. Als scheduling algorithmus wurde eine Priority-queue mit Verdrängung 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 (1,3 Punkte) (Theorie[1])

Ihre Consulting-Firma hat den Auftag bekommen, eine Software für einen Autohersteller zu entwickeln. Der Autohersteller braucht für eine neue Fabrik einen Laufbandtreiber. Das Laufband befördert die fast fertigen Autos zu einer Schraubmaschine, welche die Räder festschraubt. Dabei spielt die Koordination von Band und Schraubmaschine eine große Rolle: Damit der neue, teure Wagen nicht von der Schraubmaschine zerkratzt wird, muss das Laufband angehalten werden, wenn der Schraubarm arbeitet.

Der Treiber soll das Band anhalten, dann sollen die Schrauben in die Schraubmaschine geladen und anschließend die Schrauben festgedreht werden. Vor dem Festschrauben müssen die Bohrarme zum Auto bewegt werden und abschließend wieder wegbewegt werden. Nun kann das Laufband wieder das nächste Auto zur Maschine befördern.

Der Bohrer und das Laufband werden in dem Treiber mithilfe von jeweils einem Thread angesteuert. Die Threads sind unten dargestellt. Dein Praktikant hat eine erste Version des Treibers entwickelt. Die Logik von Band und Arm ist implementiert, die Koordination ist aber noch nicht. Die Lösung ist unten abgebildet.

Global:

```
State s;                    // Werte: 'belt' oder 'screw'
```

Beförderungsband-Thread:

```
while(1) {
        belt_move(1);    // Band nach vorne bewegen
        s = screw;
}
```

Schraubarm-Thread:

```
while(1) {
        arm_dock();      // Schraubmaschine andocken
        screw();         // Schrauben festdrehen
        arm_undock();    // Schraubmaschine abdocken
        s = belt;
}
```

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

a) Nennen Sie ein praktisches Problem, das beim Ausführen des oben genannten Programms auftreten kann.
(0,1 Punkte)

b) Was sind *Spurious Wakeups*? Und wie kann man sicher stellen das die eigentliche Bedingung erfült ist, auch in fall von Spurious wakeup?
(0,2 Punkte)

c) Verbessern Sie obiges Programm mit *signal/wait* und *mutex*, sodass Probleme und Unfälle verhindert werden. Ihre Lösung sollte auch Spurious Wakeups berücksichtigen (Hinweis die Antwort von Teilaufgabe b) weiter verwenden). Hinweis: Sie können zusätzliche globale Variablen verwenden.
(.8 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 (0,7 Punkte) (Theorie[1])

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 $t = 0$.

a) Existiert für diese Prozesse ein zulässiger Schedule? Wird das notwendige Kriterium erfüllt? **(0,1 Punkte)**

b) Wie könnte dieser aussehen? Geben Sie etwaige Leerzeiten an und markieren Sie die Hyperperiode.
(0,3 Punkte)

Tabelle 1: Prozesse

Prozesse	Dauer (D)	Periode (P)
A	1	6
B	2	6
C	1	4

c) Ist Rate-Monotonic-Scheduling (RMS) ein gültiger Schedule? Begründen Sie Ihre Antwort. **(0,1 Punkte)**

d) Was passiert, wenn zu den Prozessen ein weiterer Prozess, Prozess D mit (D=3, P=19), hinzugefügt wird? **(0,2 Punkte)**

Aufgabe 4.6: Bäckerei (3 Punkte) (Praxis^2)

In dieser Aufgabe geht es um eine Bäckerei, die sich ausschließlich auf Muffins spezialisiert hat. In der Vorgabe finden Sie ein lauffähiges Programm, welches aber nicht korrekt funktioniert. Ihre Aufgabe wird es sein die kritischen Bereiche des Programms mit Mutexen abzusichern und die Threads zu synchronisieren. In der Vorgabe befinden sich 6 C-Dateien: **bakery.c**, **management.c**, **order.c**, **supplier.c**, **warehouse.c**, **worker.c**.
Der grobe Ablauf des Programms:
Die Bäckerei hat ein Management, welches neue Aufträge aus der Datei orders.txt einließt. Diese Datei enthält den Namen des Auftraggebers sowie die Anzahl der bestellten Muffins. Das Management legt die Aufträge in dem Ringbuffer **orders_in** ab und beendet sich nachdem alle Aufträge eingelesen oder die maximale Anzahl der Aufträge (MAX_ORDERS) erreicht ist. Desweiteren besitzt die Bäckerei ein Lager (warehouse) in diesem sind die Zutaten als Integer-Werte gespeichert und repräsentieren die Menge der Zutaten in Gramm. Es gibt eine Spedition (forwarding_agent) die Lieferanten (supplier) beauftragt, die für den Zutatennachschub sorgen. In diesem Fall sind es 3 supplier_threads; für jede Zutat jeweils einen. Zuletzt gibt es noch die Bäcker (worker). Die Bäckerei erzeugt eine Anzahl an worker_threads, die in der Konstanten **num_worker_threads** angegeben ist. Die Bäcker holen sich jeweils einen Auftrag vom Management, bearbeiten diesen, indem sie aus dem Lager die benötigten Zutaten holen. Wenn sie einen Auftrag erledigt haben, legen sie den bearbeiteten Auftrag in dem Array **orders_finished** des Managements ab und holen sich einen neuen Auftrag. Wenn keine Aufträge mehr vorhanden sind, beenden die Bäcker ihre Arbeit. Danach werden die erstellten Datenstukturen der Bäckerei gelöscht. In der Funktion **management_destroy** wird der forwarding_agent_thread benachrichtigt, die Arbeit der Lieferanten anzuhalten. Abschließend wird eine Zusammenfassung der bearbeiteten Aufträge auf der Konsole ausgegeben.

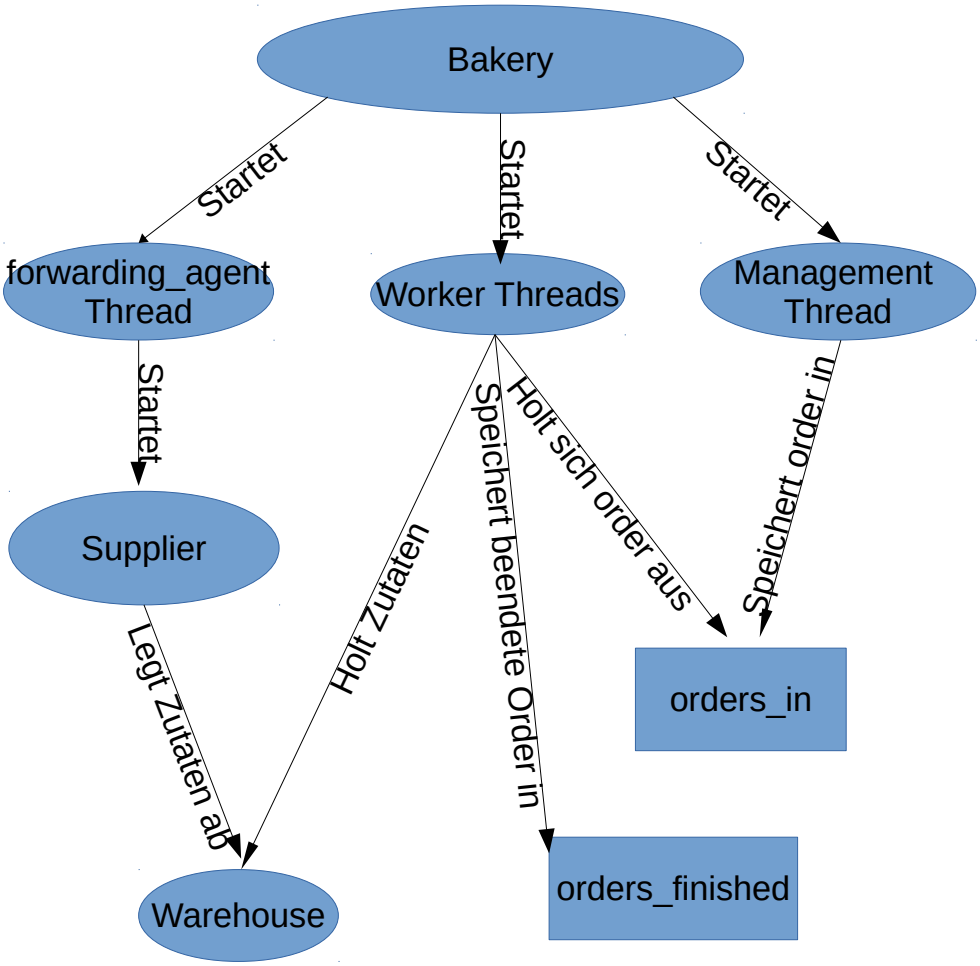


Abbildung 1: Vereinfachter Programmablauf
Für weitere Informationen zu helgrind besuchen Sie die folgende Seite:
`http://valgrind.org/docs/manual/hg-manual.html`

Hinweise:

- Verschaffen Sie sich einen Überlick über die Abläufe aller Threads
- Überlegen Sie auf welche Variablen die Threads gemeinsam zugreifen (können)
- Berücksichtigen Sie auch Spurious Wakeups
- Nutzen Sie das **Makefile** in den Vorgaben zum kompilieren
- Nutzen Sie *valgrind - -tool=helgrind ./bakery_main*, 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
- 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)*