Technische Universität Berlin Fachgebiet Komplexe und Verteilte IT-Systeme Sommersemester 2017	**Aufgabenblatt 4** zu – Systemprogrammierung – Prof. Dr. Odej Kao, Dr. Peter Janacik, Tutoren
Abgabetermin:	[1] – 18.06.2017 23:55 Uhr [2] – 25.06.2017 23:55 Uhr

Aufgabe 4.1: Periodische Prozesse (0,9 Punkte) (Theorie[1])

Ein Herzschrittmacher ist ein in der Medizin verwendetes Gerät das zur Behandlung von Patienten mit z.B. zu langsamen Herzschlägen eingesetzt wird. Das Gerät simuliert regelmäßig den Herzmuskel mit Hilfe von elektrischen Impulsen und regt diesen so zur Kontraktion an. Moderne Geräte verfügen darüber hinaus auch über eine Funktion, die Herzrhythmusstörungen aufzuzeichnen und die Intensität an die Patientenaktivität anzupassen. Dabei zeichnet Prozess A die EKG Werte auf, Prozess B berechnet die aktuelle Aktivität des Patienten und Prozess C ist die Elektrode die den Herzmuskel anregt. Die Startzeitpunkte, Perioden und die von den Prozessen auf dem Herzschrittmacher benötigte Laufzeit können Sie der Tabelle **??** entnehmen:

Tabelle 1: Prozesse

Prozesse	**Startzeitpunkt**	**Dauer**	**Periode**
A	0	2	7
B	0	1	6
C	2	2	8

Außerdem gilt, dass die Frist (Deadline) eines Prozesses seine Periode ist.

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

b) Wie könnte dieser aussehen? Geben Sie etwaige Leerzeiten an. (0,5 Punkte)

c) Was passiert, wenn zu den Prozessen ein weiterer Prozess D: (5,3,7) hinzugefügt wird? (0,2 Punkte)

Aufgabe 4.2: Synchronisation/Kooperation (1,1 Punkte) (Theorie[1])

Angelehnt an das obere Beispiel hat ein Kommilitone von Ihnen ein Programm entwickelt das die EKG Werte sowie die berechnete Aktivität in eine Datei schreibt. Dabei präsentiert er die folgende Lösung:

```
Global:
logging = fopen("log.txt", "a");
char buffer[256];

Funktion 1:
while(1) {
        getEKG(&buffer);
        fprintf(logging, "%s", &buffer);
        sleep(10);
}
```

```
Funktion 2:
while(1) {
        getAktivitaet(&buffer);
        fprintf(logging, "%s", &buffer);
        sleep(10);
}
```

a) Ihr Kommilitone möchte nun Funktion 1 und Funktion 2 jeweils in einem Thread laufen lassen. Welche Probleme könnten bei dem jetzigen Programm auftreten? (0,1 Punkte)

b) Um was für eine Prozessinteraktion handelt es sich hierbei? Begründen Sie. (0,2 Punkte)

c) Um die Probleme aus a) zu beheben schlagen Sie ihm nun vor die Operationen *signal* und *wait* einzusetzen um die beiden Threads miteinander kooperieren zu lassen. Passen Sie das Programm entsprechend an. Achten Sie dabei auf gegebenenfalls neu hinzukommende Variablen. (0,3 Punkte)

d) Was versteht man unter dem Begriff Spurious Wakeup? Erläutern Sie ihn. (0,2 Punkte)

e) Beachtet ihre Lösung von Aufgabe 4.2 c) die Problematik von Spurious Wakeups? Ändern Sie gegebenenfalls Ihren Code so, dass Spurious Wakeups berücksichtigt werden. Begründen Sie. (0,3 Punkte)

Aufgabe 4.3: 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 gleichzeitig von beiden 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();
  salatUndSo ß e();
  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 4.4: POSIX Threads (Tafelübung)

a) Recherchieren Sie kurz und erläutern Sie dann in zwei bis drei Sätzen den Begriff *Spurious Wakeup*. Geben Sie an, ob und wenn ja, an welchen Stellen es bei Ihrer Lösung aus Aufgabe 4.2 e) der zweiten Theorieaufgabe dazu kommen kann und passen Sie Ihren Code gegebenfalls so an, dass diese den korrekten Programmablauf nicht gefährden.

b) Implementieren Sie die Lösung aus der vorhergehenden Unteraufgabe unter Verwendung von POSIX Threads in C.

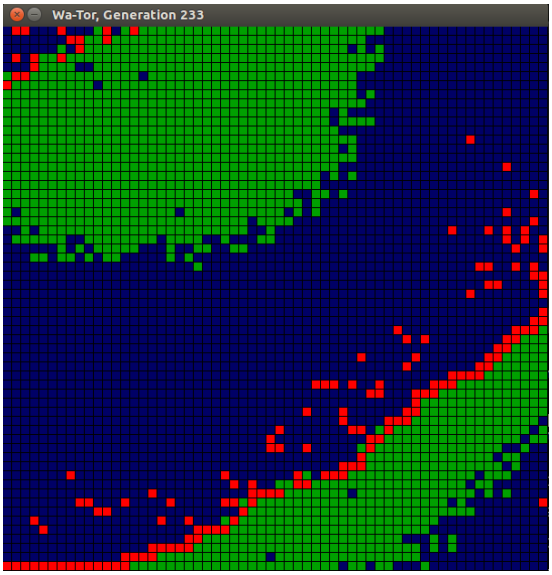
Aufgabe 4.5: Wa-Tor (3 Punkte) (Praxis[2])

In dieser Aufgabe geht es darum, eine Haifisch- und Fischpopulation in einem Aquarium zu simulieren.

Die Berechnnung erfolgt durch mehrere Threads, weshalb Ihre Aufgabe hauptsächlich darin besteht, die Berechnungs-Threads zu synchronisieren. Die Regeln für die Simulation sind bereits vorgegeben und richten sich nach den Regeln der Wa-Tor-Simulation[1].

Bereiche des Felds können verschiedene Zustände annehmen. **Rot** steht hierbei für Haie, **Grün** für Fische und **Blau** für Wasser.

Abbildung 1: Das Aquarium



Ihr Ziel ist es, die Berechnung der Population zu parallelisieren. Dies geschieht in der Code-Vorgabe durch die Berechnungs-Threads (siehe `thread.c`), die in der `main()` (siehe `wa-tor.c`) gestartet werden.

Zudem wird dort ein weiterer Thread gestartet, der das Drucken in die Konsole, oder eine GUI übernimmt. Die Kommunikation mit dem Printer-Thread ist allerdings bereits vorgegeben.

Bei der Parallelisierung der Simulation ergibt sich das Problem, das für die Berechnung der Randfelder des Zuständigkeitsbereiches jedes Threads die Zustände der benachbarten Felder benötigt werden. Da sich diese während der Berechnung ändern können, erfordert dies für jeden Randbereich Signal/Wait-Objekte.

Sie sollen **ausschließlich** in den folgenden Dateien Änderungen vornehmen und zudem ausschließlich in den mit *TODO* gekennzeichneten Stellen im Code:

```
thread.c
field.c
field.h
rules.c
```

a) Implementieren Sie folgendes in der `field.c`:

```
field* make_field(int num_threads)
```

[1] https://en.wikipedia.org/wiki/Wa-Tor

Diese Funktion erzeugt eine Instanz des Datentyps `struct field` (siehe `field.h`), initialisiert alle Variablen des Structs und sowie alle Mutex und Signal/Wait-Objekte. Das Spielfeld wird intern als Matrix des Datentyps `struct animal` repräsentiert (siehe `field.h`).

Um das Feld zu erzeugen, nutzen Sie die Funktion `population_from_file()`, wie in den Hinweisen oder in der `field.c` beschrieben.

Die Arrays aus Mutex, Signal/Wait-Objekten und Booleans die Lese- und Schreiboperationen von Reihen an Threadrändern schützen sollen, sind bereits initialisiert.

Machen Sie sich Gedanken, welche Mutexe, Signal/Wait-Objekte oder ggf. weitere Variablen Sie benötigen, um die Synchronisation aus der zweiten Aufgabe zu realisieren und deklarieren Sie sie hier. Erweitern Sie dazu ebenfalls die `field.h` um diese Objekte. **Beachten Sie weitere Hinweise im Code.**
(0,8 Punkte)

b) Implementieren Sie folgendes in der `thread.c`:

```
void* thread(void* t_args)
```

Diese Funktion wird von jedem Thread aufgerufen.

In einer Dauerschleife wird hier gewartet, bis der Drucker-Thread die letzte Generation ausgegeben hat, bevor die nächste Generation des Zuständigkeitsbereiches des jeweiligen Threads berechnet wird.

Ihre Aufgabe ist es, die Generationszahl aus dem `struct field` zu inkrementieren.

Des Weiteren sollen Sie sicher stellen, dass alle Threads in ihren Bereich des Felds geschrieben haben, also die Funtionen `calculate_shark_generation()` und `calculate_fish_generation()` aufgerufen haben, bevor dem Drucker-Thread signalisiert wird, die neue Generation auszudrucken.

Nutzen Sie hierbei die von Ihnen hinzugefügten Signal/Wait-Objekte aus der vorherigen Aufgabe. **Beachten Sie weitere Hinweise im Code.**
(1,2 Punkte)

c) Implementieren Sie in der `rules.c` folgende Funktionen:

```
void lock(thread_args* args, int h)
void unlock(thread_args* args, int h)
```

In der `rules.c` erfolgt die eigentliche Berechnung der Folgegeneration. Diese unterteilt sich in `calculate_shark_generation` und `calculate_fish_generation`, die in der `thread.c` nacheinander aufgerufen werden.

Da in diesen Funktionen sehr oft Reihen an den Rändern des Zuständigkeitsbereiches gelesen oder modifiziert werden, ist es sinnvoll, das Sperren und Entsperren dieser Reihen auszulagern. Ihre Aufgabe ist es, diese Funktionen zu implementieren.

In den Funktionsrümpfen sind bereits Pointer auf die benötigten Signal/Wait-Objekte deklariert.

Nutzen Sie die Funktion `to_lock()` um die bereits erwähnten Signal/Wait-Objekte zu initialisieren. Achten Sie darauf, dass `to_lock()` die Objekte mit `NULL` initialisiert, falls die Reihe nicht ge- bzw. entsperrt werden muss.

In der Funktion `lock()` müssen Sie weiter überprüfen, ob die Objekte bereits gesperrt sind und gegebenenfalls auf deren Freigabe warten.

In der Funktion `unlock()` müssen Sie den wartenden Threads signalisieren, dass die Reihe entsperrt wurde.

Beachten Sie weitere Hinweise im Code.
(1 Punkt)

Hinweise:

- Die Funktion `population_from_file(int* height, int* width)` liest eine Datei mit der Startpopulation ein, deren Pfad in der `config.h` angegeben ist. Der Rückgabewert ist dabei die Matrix aus `struct animal` Einträgen. Damit die Einträge `int height` und `int width` aus dem `struct field` richtig nach der in der Datei beschriebenen Größe angepasst werden, werden ihre Adressen hierbei der Funktion übergeben. Der Speicher, der für die `struct animal` Matrix benötigt wird, wird in der Funktion alloziert.
- Arbeiten Sie für den Anfang nur mit *einem* Berechnungs-Thread.
- Sie dürfen weitere Funktionen in der `rules.c` implementieren, um die mit *TODO* gekennzeichneten Aufgaben zu lösen.
- Sie können Ihr Spiel auch mit anderen Eingaben testen um die Balance zwischen Haien und Fischen zu beeinflussen. Die `config.c` bietet viele Möglichkeiten der Anpassung, unter anderem die Simulationsgeschwindigkeit, die Überlebensdauer der Haie ohne Futter, die Fortpflanzungsrate der Fische und Haie usw. Sollten Sie die Konsolenausgabe des Spielfelds benutzen (`GUI = 0`), achten Sie darauf, dass Sie die Größe der Konsole anpassen, um das komplette Spielfeld zu sehen, oder ggf. ein kleineres Startset in der `config.h` anzugeben.
- Sie können in der `config.h` den Pfad zu der Datei mit der Startpopulation ändern. Diese befindet sich im Unterordner `population_sets`. Achten Sie hierbei auf das richtige Format, das in der Textdatei im Ordner beschrieben ist.
- Sie können das mitgelieferte `Makefile` zum Kompilieren und Ausführen benutzen:
  - `make all`: Kompiliert alle relevanten Dateien
  - `make run`: Führt das Programm aus (Abbruch über *STRG + C*)
  - `make debug`: Führt das Programm zum Debuggen mit `gdb` aus
  - `make clean`: Löscht beim Kompilieren erstellte Objektdateien
- In der Code-Vorgabe wird an vielen Stelle **einmalig** dynamischer Speicher alloziert. Da das Programm in einer Dauerschleife läuft und mit *STRG + C* beendet wird, ist es nicht nötig diesen im Code wieder freizugeben - das geschieht beim Beenden des Prozesses automatisch.