Musterlösung der Probeklausur

1. Synchronisation (10/52 Punkte)

a) Was ist ein **Mutex**?

Ein Mutex (mutual exclusion) ist eine boolsche Variable, die man nutzen kann, um Betreten und Verlassen eines kritischen Bereichs zu steuern: Nur ein Prozess/Thread kann den Wert von true auf false setzen; weitere, die dies ebenfalls versuchen, müssen warten.

b) Der Linux-Kernel verwendet **Spin Lock** genannte Mutexe. Diese Spin Locks legen sich nicht schlafen. Inwiefern hat das Auswirkungen auf die Verwendbarkeit innerhalb von Interrupt-Handlern?

Da sie nicht schlafen, darf man sie in Interrupt-Handlern benutzen – denn die dürfen nicht schlafen (weil es keine Möglichkeit gäbe, sie wieder aufzuwecken: es sind ja keine Prozesse).

c) Beschreiben Sie das **Erzeuger-Verbraucher-Problem** und skizzieren Sie mit Hilfe von Pseudo-Programm-Code eine Semaphor-basierte Lösung. (Verwenden Sie wahlweise eine an C oder an Python erinnernde Syntax; exakte Befehlsnamen etc. sind irrelevant.)

Ein Erzeuger stellt Objekte in einen beschränkten Puffer, aus dem ein Verbraucher sie liest. Der Puffer darf nicht „übervoll" geschrieben werden, und der Verbraucher darf nicht aus einem leeren Puffer lesen.

Code:

```
int MAX = 4;
semaphor mutex = 1;
semaphor sem_read = 0;
semaphor sem_write = MAX;

// Erzeuger:
loop forever {
   wait (sem_write);
   wait (mutex);
   in_puffer_schreiben ();
   signal (mutex);
   signal (sem_read);
}

// Verbraucher:
loop forever {
   wait (sem_read);
   wait (mutex);
   aus_puffer_lesen ();
   signal (mutex);
   signal (sem_write);
}
```

2. Prozesse und Threads (8/52 Punkte)

a) Linux verwendet das **POSIX-Thread-Modell** für die Thread-Programmierung. Schreiben Sie in Pseudo-Code (darf wie ein C- oder wie Python-Programm „aussehen") ein kleines Programm, das zwei Threads erzeugt, die beide „Hello World" ausgeben. Das Programm soll erst beendet werden, wenn beide Threads beendet sind.


```
void * hello_world () { printf ("Hello World"); };

int main () {
  p_thread t1, t2;
  pthread_create (t1, NULL, &hello_world, NULL);
  pthread_create (t2, NULL, &hello_world, NULL);
  pthread_join (t1);
  pthread_join (t2);
}
```

b) Wie unterscheiden sich allgemein **User-** und **Kernel-Level-Threads** und welche Vor- und Nachteile sind damit jeweils verbunden?

- KLTs werden vom Kernel verwaltet, ULTs von einer User-Level-Bibliothek, bei ULTs weiß der Kernel nicht, dass es Threads gibt

- Vorteil ULTs: besonders schneller Thread-Wechsel (kein Umschalten in den Kernel-Modus nötig)

- Vorteil KLTs: Da Kernel die Threads kennt, kann er, wenn ein Thread mit I/O blockiert, einen anderen Thread desselben Prozesses ausführen. Bei ULTs blockiert der ganze Prozess.

3. System calls (10/52 Punkte)

a) Nach dem Öffnen einer Datei mit **`fd=open(...)`** wollen Sie lesend und schreibend auf die Datei zugreifen. Geben Sie die Syntax der Befehle zum Lesen und Schreiben an, d. h. nennen Sie die Parameter der beiden Funktionen und erklären Sie die Bedeutungen der Parameter. (Die korrekte Reihenfolge der möglichen Parameter ist dabei nicht wichtig.)

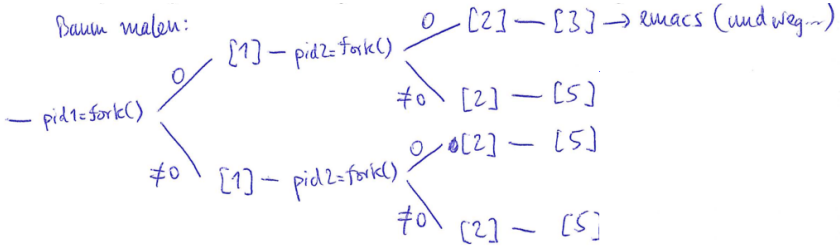
```
int buffer[20];
fd = open ("test.dat", O_RDWR);
read (fd, &buffer, 20);
write (fd, &buffer, 20);
```

fd ist der File-Descriptor, der aus open() kommt, buffer ist der Puffer, der beim Lesen die Daten aufnimmt bzw. sie beim Schreiben enthält, das letzte Argument bei read/write ist die Anzahl Bytes, die die Funktion lesen/schreiben soll.

b) Betrachten Sie den folgenden Programmausschnitt:

```
int pid1 = fork();
printf ("%s\n","[1] Ein Fork ist durch, einer muss noch.");
int pid2 = fork();
printf ("%s\n","[2] Zeit für eine Fallunterscheidung.");
if ( (pid1==0) && (pid2==0) ) {
  printf ("%s\n","[3] Ich starte jetzt emacs.");
  execl ("/bin/emacs", "/etc/fstab", (char *)NULL);
  int pid3 = fork();
  printf ("%s\n","[4] Nach dem dritten Fork.");
} else {
  printf ("%s\n","[5] Ich gucke nur zu.");
};
```

Wie oft und warum erscheinen die mit [1] bis [5] durchnummerierten Ausgaben? Schreiben Sie zu jeder Ausgabe die Anzahl und begründen Sie Ihre Antwort stichwortartig oder mit einer Skizze.



c) Beim Aufruf des System calls **`fork()`** erhält der Sohn den Wert 0 und der Vater die Prozess-ID des neu erzeugten Sohnes (≠0) zurück. Für eine reine Unterscheidung („wer bin ich?“) könnte man auch umgekehrt arbeiten, also im Vater den Wert 0 und im Sohn die Prozess-ID des Vaters (≠0) zurückgeben. Warum ist diese Alternative schlecht?

Der Sohn kann immer über getppid() (get parent process id) die ID des Vaterprozesses rausfinden, denn da gibt es ja nur einen. Anders rum kann aber der Vater nicht die PID eines bestimmten Sohnes über einen Funktionsaufruf erfahren – es könnte ja mehrere geben.

4. Scheduling-Verfahren (Uni-Prozessor) (11/52 Punkte)

a) Aus der Vorlesung kennen Sie die Scheduling-Verfahren **FCFS** (First Come First Served), **SJF** (Shortest Job First) und Round Robin (**RR**).

Es gebe die folgenden fünf Prozesse mit den angegebenenen Ankunftszeiten und Gesamtrechenzeiten:

Prozess	Ankunft	Rechenzeit
P	0	10
Q	4	8
R	5	7
S	6	1
T	12	2

Für First Come First Served sieht die Ausführreihenfolge wie folgt aus:

Zeit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
FCFS	**P**	P	P	P	P	P	P	P	P	P	**Q**	Q	Q	Q	Q	Q	Q	Q	**R**	R	R	R	R	R	**S**	**T**	T	
SJF	**P**	P	P	P	P	P	P	P	P	P	**S**	**R**	R	R	R	R	R	R	**T**	T	**Q**	Q	Q	Q	Q	Q	Q	Q
RR (q=4)	**P**	P	P	P	**Q**	Q	Q	Q	**P**	P	P	P	**R**	R	R	R	S	**Q**	Q	Q	Q	**T**	T	**P**	P	**R**	R	R
RR (q=10)	**P**	P	P	P	P	P	P	P	P	P	**Q**	Q	Q	Q	Q	Q	Q	Q	**R**	R	R	R	R	R	**S**	**T**	T	

Ergänzen Sie für SJF und RR hier in der Tabelle die Ausführreihenfolgen. Für RR nehmen Sie ein Zeitquantum von q=4 bzw. q=10 Zeiteinheiten an. (Neue Jobs werden bei RR im Moment ihrer Ankunft hinten an die aktuelle Warteschlange angehängt.)

q=4:

t=0: P(10) auswählen, Queue = [ ]

t=4: Q kommt hinzu: Queue = [Q(8)]
P(6) unterbrechen, an Queue hängen, Queue = [Q(8), P(6)]
Q(8) auswählen und laufen lassen, Queue = [P(6)]

t=5: R kommt hinzu: Queue = [P(6), R(7)]

t=6: S kommt hinzu: Queue = [P(6), R(7), S(1)]

t=8: Q(4) unterbrechen, an Queue anhängen, Queue = [P(6), R(7), S(1), Q(4)]
P(6) auswählen und laufen lassen, Queue = [R(7), S(1), Q(4)]

t=12: T kommt hinzu: Queue = [R(7), S(1), Q(4), T(2)]
P(2) unterbrechen und an Queue anhängen, Queue = [R(7), S(1), Q(4), T(2), P(2)]
R(7) auswählen und laufen lassen, Queue = [S(1), Q(4), T(2), P(2)]

t=16: R(3) unterbrechen und an Queue anhängen, Queue = [S(1), Q(4), T(2), P(2), R(3)]
S(1) auswählen und laufen lassen, Queue = [Q(4), T(2), P(2), R(3)]

t=17: S(0) ist fertig, entfernen
Q(4) auswählen und laufen lassen, Queue = [T(2), P(2), R(3)]

t=21: Q(0) ist fertig, entfernen
T(2) auswählen und laufen lassen, Queue = [P(2), R(3)]

t=23: T(0) ist fertig, entfernen
P(2) auswählen und laufen lassen, Queue = [R(3)]

t=25: P(0) ist fertig, entfernen
R(3) auswählen und laufen lassen, Queue = [ ]

t=28: R(0) ist fertig, entfernen, Queue leer, Ende.

b) Wenn Sie FCFS mit RR(q=4) und RR(q=10) vergleichen, was fällt Ihnen dann auf? Bewerten Sie die Wahl des Zeitquantums q=4 bzw. q=10.

RR(q=10) verhält sich (hier) wie FCFS, weil kein Prozess länger als 10 Zeiteinheiten läuft. Für das Beispiel ist das Quantum 10 also zu lang.

c) Was ist der Unterschied zwischen **I/O-lastigen Prozessen** und **CPU-lastigen Prozessen**?

I/O-lastig: verbringt die meiste Zeit mit Warten auf I/O-Operationen

CPU-lastig: rechnet die meiste Zeit

5. Deadlocks (4/52 Punkte)

a) Auf einem System gebe es fünf Prozesse P_1, P_2, ..., P_5 und fünf exklusive Betriebsmittel R_1, R_2, ..., R_5. Der momentane Zustand sei wie folgt:

P_1 belegt keine Ressource und fordert R_1 an,

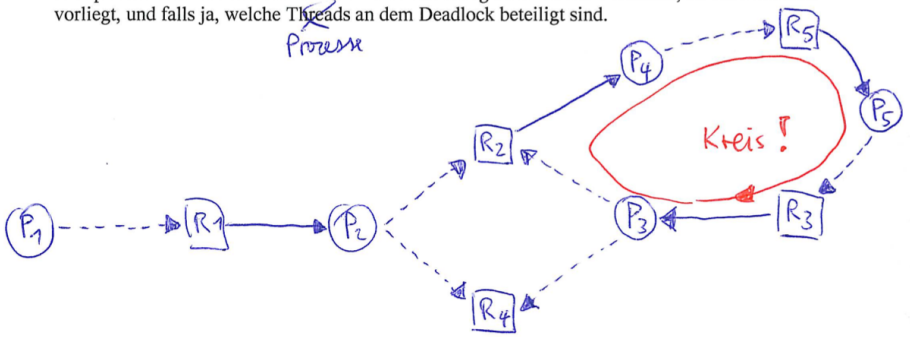
P_2 belegt R_1 und fordert R_2 und R_4 an,

P_3 belegt R_3 und fordert R_2 und R_4 an,

P_4 belegt R_2 und fordert R_5 an,

P_5 belegt R_5 und fordert R_3 an.

Überprüfen Sie mit einem der beiden in der Vorlesung behandelten Verfahren, ob ein Deadlock vorliegt, und falls ja, welche ~~Threads~~ Prozesse an dem Deadlock beteiligt sind.



Deadlocks der Prozesse P_3, P_4, P_5

6. Interrupts (9/52 Punkte)

a) Nennen Sie – im Umgang mit an einen Computer angeschlossener Hardware – eine **Alternative zur Verwendung von Interrupts**, und erläutern Sie, warum Interrupts deutliche Verbesserungen bei Performance und Einfachheit der Programmierung liefern.

Die Alternative heißt Polling, dabei wird in einer Schleife immer wieder der Zustand eines Geräts abgefragt. Im Vergleich zur Arbeit mit Interrupts verbraucht das viel sinnlose Rechenzeit, und auch die Programmlogik wird umständlicher, weil sich hier das Programm bewusst ist, dass es auf ein I/O-Ereignis wartet.

b) Während der Bearbeitung eines Interrupts kommt es zu einem **weiteren Interrupt**. Bei der Implementierung eines Betriebssystems haben Sie verschiedene Möglichkeiten, eine solche Situation zu behandeln. Nennen Sie zwei davon und erläutern Sie Vor- und Nachteile.

(1) Während der Interrupt-Behandlung sind alle anderen Interrupts gesperrt.

(2) Während der Interrupt-Behandlung sind Interrupts weiter zugelassen, sie unterbrechen die aktuell laufende Interrupt-Behandlung

(3) Es gibt Interrupt-Prioritäten: Nur wenn ein Interrupt auftritt, der höhere Priorität als der gerade behandelte hat, wird die laufende IRQ-Behandlung unterbrochen

Vorteil (1) gegenüber (2): man verpasst keine Interrupts

Nachteil (1) gegenüber (2): Die Bearbeitung eines wichtigen Interrupts kann immer wieder verzögert werden, weil ständig neue (aber weniger wichtige) Interrupts reinkommen.

c) Linux teilt Interrupt-Behandlungsroutinen (Interrupt Handler) in zwei Teile auf, eine **top half** und eine **bottom half** (auch **Tasklet** genannt). Wie unterscheiden sich top und bottom half, und warum führt man diese Trennung ein?

Die top half ist der eigentliche Interrupt Handler, der vom OS beim Auftreten eines Interrupts aktiviert wird. Er führt nur die wichtigsten (zeitkritischen) Dinge durch, etwa Abfragen eines Gerätestatus und Rücksetzen des Geräts („Rückmeldung: Ich habe Deinen Interrupt gesehen und bearbeitet").

Die bottom half (Tasklet) führt die restlichen Schritte durch, die nicht zeitkritisch sind und darum nicht in der top half stattfinden.

Der Sinn der Trennung ist, möglichst schnell den eigentlichen Interrupt Handler zu beenden, um (im Fall der Deaktivierung von weiteren Interrupts) Interrupts wieder zuzulassen.